JavaScript®

Notes for Professionals

400+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/JavaScriptBook

This *JavaScript® Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official JavaScript® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with JavaScript

Version	Release Date
1	1997-06-01
2	1998-06-01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01
5.1	2011-06-01
6	2015-06-01
7	2016-06-14
8	2017-06-27

Section 1.1: Using console.log()

Introduction

All modern web browsers, Node.js as well as almost every other JavaScript environments support writing messages to a console using a suite of logging methods. The most common of these methods is `console.log()`.

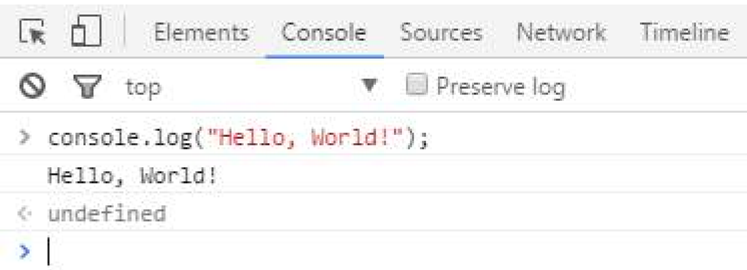
In a browser environment, the `console.log()` function is predominantly used for debugging purposes.

Getting Started

Open up the JavaScript Console in your browser, type the following, and press `Enter`:

```
console.log("Hello, World!");
```

This will log the following to the console:



In the example above, the `console.log()` function prints `Hello, World!` to the console and returns **`undefined`** (shown above in the console output window). This is because `console.log()` has no explicit *return value*.

Logging variables

`console.log()` can be used to log variables of any kind; not only strings. Just pass in the variable that you want to be displayed in the console, for example:

```
var foo = "bar";
console.log(foo);
```

This will log the following to the console:

```
> var foo = "bar";
  console.log(foo);
  bar
< undefined
```

If you want to log two or more values, simply separate them with commas. Spaces will be automatically added between each argument during concatenation:

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

This will log the following to the console:

```
> var thisVar = 'first value';
  var thatVar = 'second value';
  console.log("thisVar:", thisVar, "and thatVar:", thatVar);
  thisVar: first value and thatVar: second value
< undefined
```

Placeholders

You can use `console.log()` in combination with placeholders:

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

This will log the following to the console:

```
> var greet = "Hello", who = "World";
  console.log("%s, %s!", greet, who);
  Hello, World!
< undefined
```

Logging Objects

Below we see the result of logging an object. This is often useful for logging JSON responses from API calls.

```
console.log({
    'Email': '',
    'Groups': {},
    'Id': 33,
    'IsHiddenInUI': false,
    'IsSiteAdmin': false,
    'LoginName': 'i:0#.w|virtualdomain\\user2',
    'PrincipalType': 1,
```

```
    'Title': 'user2'
});
```

This will log the following to the console:

```
▼Object {Email: "", Groups: Object, Id: 33, IsHiddenInUI: false, IsSiteAdmin: false…}
   Email: ""
 ▶Groups: Object
   Id: 33
   IsHiddenInUI: false
   IsSiteAdmin: false
   LoginName: "i:0#.w|virtualdomain\user2"
   PrincipalType: 1
   Title: "user2"
 ▶__proto__: Object
```

Logging HTML elements

You have the ability to log any element which exists within the *DOM*. In this case we log the body element:

```
console.log(document.body);
```

This will log the following to the console:

```
▼<body class="question-page new-topbar">
    <noscript><div id="noscript-padding"></div></noscript>
    <div id="notify-container"></div>
    <div id="custom-header"></div>
  ▶<header class="so-header js-so-header _fixed">…</header>
  ▶<script>…</script>
  ▶<div class="container">…</div>
    <script async src="https://cdn.sstatic.net/clc/clc.min.js?v=51f344c0b478"></script>
  ▶<div id="footer" class="categories">…</div>
  ▶<noscript>…</noscript>
  ▶<script>…</script>
  ▶<script>…</script>
  ▶<script>…</script>
  ▶<script type="text/javascript">…</script>
 </body>
```

End Note

For more information on the capabilities of the console, see the Console topic.

Section 1.2: Using the DOM API

DOM stands for **D**ocument **O**bject **M**odel. It is an object-oriented representation of structured documents like XML and HTML.

Setting the `textContent` property of an `Element` is one way to output text on a web page.

For example, consider the following HTML tag:

```
<p id="paragraph"></p>
```

To change its `textContent` property, we can run the following JavaScript:

```
document.getElementById("paragraph").textContent = "Hello, World";
```

This will select the element that with the id `paragraph` and set its text content to "Hello, World":

```
<p id="paragraph">Hello, World</p>
```

(See also this demo)

You can also use JavaScript to create a new HTML element programmatically. For example, consider an HTML document with the following body:

```
<body>
   <h1>Adding an element</h1>
</body>
```

In our JavaScript, we create a new **`<p>`** tag with a `textContent` property of and add it at the end of the html body:

```
var element = document.createElement('p');
element.textContent = "Hello, World";
document.body.appendChild(element); //add the newly created element to the DOM
```

That will change your HTML body to the following:

```
<body>
   <h1>Adding an element</h1>
   <p>Hello, World</p>
</body>
```

Note that in order to manipulate elements in the DOM using JavaScript, the JavaScript code must be run *after* the relevant element has been created in the document. This can be achieved by putting the JavaScript **`<script>`** tags *after* all of your other **`<body>`** content. Alternatively, you can also use an event listener to listen to eg. `window's onload event`, adding your code to that event listener will delay running your code until after the whole content on your page has been loaded.

A third way to make sure all your DOM has been loaded, is to wrap the DOM manipulation code with a timeout function of 0 ms. This way, this JavaScript code is re-queued at the end of the execution queue, which gives the browser a chance to finish doing some non-JavaScript things that have been waiting to finish before attending to this new piece of JavaScript.

Section 1.3: Using window.alert()

The `alert` method displays a visual alert box on screen. The alert method parameter is displayed to the user in **plain** text:

```
window.alert(message);
```

Because `window` is the global object, you can call also use the following shorthand:

```
alert(message);
```

So what does `window.alert()` do? Well, let's take the following example:

```
alert('hello, world');
```

In Chrome, that would produce a pop-up like this:

Notes

> The `alert` method is technically a property of `window` object, but since all `window` properties are automatically global variables, we can use `alert` as a global variable instead of as a property of `window` - meaning you can directly use `alert()` instead of `window.alert()`.

Unlike using `console.log`, `alert` acts as a modal prompt meaning that the code calling `alert` will pause until the prompt is answered. Traditionally this means that *no other JavaScript code will execute* until the alert is dismissed:

```
alert('Pause!');
console.log('Alert was dismissed');
```

However the specification actually allows other event-triggered code to continue to execute even though a modal dialog is still being shown. In such implementations, it is possible for other code to run while the modal dialog is being shown.

More information about usage of the `alert` method can be found in the modals prompts topic.

The use of alerts is usually discouraged in favour of other methods that do not block users from interacting with the page - in order to create a better user experience. Nevertheless, it can be useful for debugging.

Starting with Chrome 46.0, `window.alert()` is blocked inside an **<iframe>** unless its sandbox attribute has the value allow-modal.

Section 1.4: Using window.prompt()

An easy way to get an input from a user is by using the `prompt()` method.

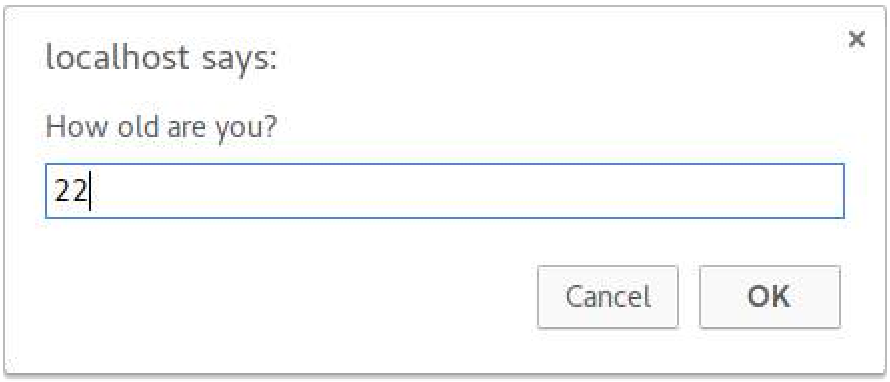
Syntax

```
prompt(text, [default]);
```

- **text**: The text displayed in the prompt box.
- **default**: A default value for the input field (optional).

Examples

```
var age = prompt("How old are you?");
console.log(age); // Prints the value inserted by the user
```



If the user clicks the OK button, the input value is returned. Otherwise, the method returns `null`.

The return value of `prompt` is always a string, unless the user clicks `Cancel`, in which that case it returns `null`. Safari is an exception in that when the user clicks Cancel, the function returns an empty string. From there, you can convert the return value to another type, such as an integer.

Notes

- While the prompt box is displayed, the user is prevented from accessing other parts of the page, since dialog boxes are modal windows.
- Starting with Chrome 46.0 this method is blocked inside an **`<iframe>`** unless its sandbox attribute has the value allow-modal.

Section 1.5: Using window.confirm()

The `window.confirm()` method displays a modal dialog with an optional message and two buttons, OK and Cancel.

Now, let's take the following example:

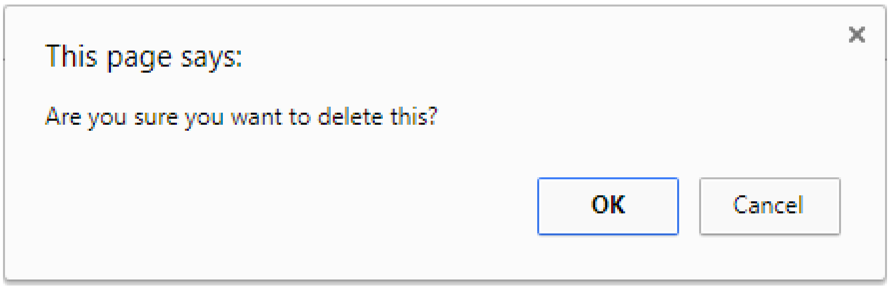
```
result = window.confirm(message);
```

Here, **message** is the optional string to be displayed in the dialog and **result** is a boolean value indicating whether OK or Cancel was selected (true means OK).

`window.confirm()` is typically used to ask for user confirmation before doing a dangerous operation like deleting something in a Control Panel:

```
if(window.confirm("Are you sure you want to delete this?")) {
    deleteItem(itemId);
}
```

The output of that code would look like this in the browser:



If you need it for later use, you can simply store the result of the user's interaction in a variable:

```
var deleteConfirm = window.confirm("Are you sure you want to delete this?");
```

Notes

- The argument is optional and not required by the specification.
- Dialog boxes are modal windows - they prevent the user from accessing the rest of the program's interface until the dialog box is closed. For this reason, you should not overuse any function that creates a dialog box (or modal window). And regardless, there are very good reasons to avoid using dialog boxes for confirmation.
- Starting with Chrome 46.0 this method is blocked inside an `<iframe>` unless its sandbox attribute has the value allow-modal.
- It is commonly accepted to call the confirm method with the window notation removed as the window object is always implicit. However, it is recommended to explicitly define the window object as expected behavior may change due to implementation at a lower scope level with similarly named methods.

Section 1.6: Using the DOM API (with graphical text: Canvas, SVG, or image file)

Using canvas elements

HTML provides the canvas element for building raster-based images.

First build a canvas for holding image pixel information.

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

Then select a context for the canvas, in this case two-dimensional:

```
var ctx = canvas.getContext('2d');
```

Then set properties related to the text:

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

Then insert the `canvas` element into the page to take effect:

```
document.body.appendChild(canvas);
```

Using SVG

SVG is for building scalable vector-based graphics and can be used within HTML.

First create an SVG element container with dimensions:

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

Then build a `text` element with the desired positioning and font characteristics:

```
var text = document.createElementNS('http://www.w3.org/2000/svg', 'text');
text.setAttribute('x', '0');
```

```
text.setAttribute('y', '50');
text.style.fontFamily = 'Times New Roman';
text.style.fontSize = '50';
```

Then add the actual text to display to the `text`element:

```
text.textContent = 'Hello world!';
```

Finally add the `text` element to our `svg` container and add the `svg` container element to the HTML document:

```
svg.appendChild(text);
document.body.appendChild(svg);
```

Image file

If you already have an image file containing the desired text and have it placed on a server, you can add the URL of the image and then add the image to the document as follows:

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

Chapter 2: JavaScript Variables

variable_name **{Required} The name of the variable: used when calling it.**
= **[Optional]** Assignment (defining the variable)
value **{Required when using Assignment}** The value of a variable **[default: undefined]**

Variables are what make up most of JavaScript. These variables make up things from numbers to objects, which are all over JavaScript to make one's life much easier.

Section 2.1: Defining a Variable

```
var myVariable = "This is a variable!";
```

This is an example of defining variables. This variable is called a "string" because it has ASCII characters (`A-Z`, `0-9`, `!@#$`, etc.)

Section 2.2: Using a Variable

```
var number1 = 5;
number1 = 3;
```

Here, we defined a number called "number1" which was equal to 5. However, on the second line, we changed the value to 3. To show the value of a variable, we log it to the console or use `window.alert()`:

```
console.log(number1); // 3
window.alert(number1); // 3
```

To add, subtract, multiply, divide, etc., we do like so:

```
number1 = number1 + 5; // 3 + 5 = 8
number1 = number1 - 6; // 8 - 6 = 2
var number2 = number1 * 10; // 2 (times) 10 = 20
var number3 = number2 / number1; // 20 (divided by) 2 = 10;
```

We can also add strings which will concatenate them, or put them together. For example:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Section 2.3: Types of Variables

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)
var myLong = 9310141419482; // 64-bit number (from -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807)
var myFloat = 5.5; // 32-bit floating-point number (decimal)
var myDouble = 9310141419482.22; // 64-bit floating-point number

var myBoolean = true; // 1-bit true/false (0 or 1)
var myBoolean2 = false;

var myNotANumber = NaN;
var NaN_Example = 0/0; // NaN: Division by Zero is not possible

var notDefined; // undefined: we didn't define it to anything yet
window.alert(aRandomVariable); // undefined
```

```
var myNull = null; // null
// etc...
```

Section 2.4: Arrays and Objects

```
var myArray = []; // empty array
```

An array is a set of variables. For example:

```
var favoriteFruits = ["apple", "orange", "strawberry"];
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];
var employees = ["Billy", "Bob", "Joe"];
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
var randomVariables = [2, "any type works", undefined, null, true, 2.51];

myArray = ["zero", "one", "two"];
window.alert(myArray[0]); // 0 is the first element of an array
                          // in this case, the value would be "zero"
myArray = ["John Doe", "Billy"];
elementNumber = 1;

window.alert(myArray[elementNumber]); // Billy
```

An object is a group of values; unlike arrays, we can do something better than them:

```
myObject = {};
john = {firstname: "John", lastname: "Doe", fullname: "John Doe"};
billy = {
    firstname: "Billy",
    lastname: undefined,
    fullname: "Billy"
};
window.alert(john.fullname); // John Doe
window.alert(billy.firstname); // Billy
```

Rather than making an array `["John Doe", "Billy"]` and calling `myArray[0]`, we can just call `john.fullname` and `billy.fullname`.

Chapter 3: Built-in Constants

Section 3.1: null

null is used for representing the intentional absence of an object value and is a primitive value. Unlike **undefined**, it is not a property of the global object.

It is equal to **undefined** but not identical to it.

```
null == undefined; // true
null === undefined; // false
```

CAREFUL: The **typeof null** is `'object'`.

```
typeof null; // 'object';
```

To properly check if a value is **null**, compare it with the strict equality operator

```
var a = null;

a === null; // true
```

Section 3.2: Testing for NaN using isNaN()

`window.isNaN()`

The global function `isNaN()` can be used to check if a certain value or expression evaluates to **NaN**. This function (in short) first checks if the value is a number, if not tries to convert it (*), and then checks if the resulting value is **NaN**. For this reason, **this testing method may cause confusion**.

(*) The "conversion" method is not that simple, see ECMA-262 18.2.3 for a detailed explanation of the algorithm.

These examples will help you better understand the `isNaN()` behavior:

```
isNaN(NaN);          // true
isNaN(1);            // false: 1 is a number
isNaN(-2e-4);        // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity);     // false: Infinity is a number
isNaN(true);         // false: converted to 1, which is a number
isNaN(false);        // false: converted to 0, which is a number
isNaN(null);         // false: converted to 0, which is a number
isNaN("");           // false: converted to 0, which is a number
isNaN(" ");          // false: converted to 0, which is a number
isNaN("45.3");       // false: string representing a number, converted to 45.3
isNaN("1.2e3");      // false: string representing a number, converted to 1.2e3
isNaN("Infinity");   // false: string representing a number, converted to Infinity
isNaN(new Date);     // false: Date object, converted to milliseconds since epoch
isNaN("10$");        // true : conversion fails, the dollar sign is not a digit
isNaN("hello");      // true : conversion fails, no digits at all
isNaN(undefined);    // true : converted to NaN
isNaN();             // true : converted to NaN (implicitly undefined)
isNaN(function(){}); // true : conversion fails
isNaN({});           // true : conversion fails
isNaN([1, 2]);       // true : converted to "1, 2", which can't be converted to a number
```

This last one is a bit tricky: checking if an `Array` is **NaN**. To do this, the `Number()` constructor first converts the array to a string, then to a number; this is the reason why `isNaN([])` and `isNaN([34])` both return **`false`**, but `isNaN([1, 2])` and `isNaN([true])` both return **`true`**: because they get converted to `""`, `"34"`, `"1,2"` and `"true"` respectively. In general, **an array is considered `NaN` by `isNaN()` unless it only holds one element whose string representation can be converted to a valid number**.

Version ≥ 6
`Number.isNaN()`

In ECMAScript 6, the `Number.isNaN()` function has been implemented primarily to avoid the problem of `window.isNaN()` of forcefully converting the parameter to a number. `Number.isNaN()`, indeed, **doesn't try to convert** the value to a number before testing. This also means that **only values of the type number, that are also `NaN`, return `true`** (which basically means only `Number.isNaN(NaN)`).

From ECMA-262 20.1.2.4:

> When the `Number.isNaN` is called with one argument `number`, the following steps are taken:
>
> 1. If Type(number) is not Number, return **`false`**.
> 2. If number is **`NaN`**, return **`true`**.
> 3. Otherwise, return **`false`**.

Some examples:

```
// The one and only
Number.isNaN(NaN);          // true

// Numbers
Number.isNaN(1);            // false
Number.isNaN(-2e-4);        // false
Number.isNaN(Infinity);     // false

// Values not of type number
Number.isNaN(true);         // false
Number.isNaN(false);        // false
Number.isNaN(null);         // false
Number.isNaN("");           // false
Number.isNaN(" ");          // false
Number.isNaN("45.3");       // false
Number.isNaN("1.2e3");      // false
Number.isNaN("Infinity");   // false
Number.isNaN(new Date);     // false
Number.isNaN("10$");        // false
Number.isNaN("hello");      // false
Number.isNaN(undefined);    // false
Number.isNaN();             // false
Number.isNaN(function(){}); // false
Number.isNaN({});           // false
Number.isNaN([]);           // false
Number.isNaN([1]);          // false
Number.isNaN([1, 2]);       // false
Number.isNaN([true]);       // false
```

Section 3.3: NaN

`NaN` stands for "Not a Number." When a mathematical function or operation in JavaScript cannot return a specific number, it returns the value **`NaN`** instead.

It is a property of the global object, and a reference to [Number.NaN](#)

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Perhaps confusingly, `NaN` is still considered a number.

```
typeof NaN; // 'number'
```

Don't check for `NaN` using the equality operator. See `isNaN` instead.

```
NaN == NaN  // false
NaN === NaN // false
```

Section 3.4: undefined and null

At first glance it may appear that `null` and `undefined` are basically the same, however there are subtle but important differences.

`undefined` is the absence of a value in the compiler, because where it should be a value, there hasn't been put one, like the case of an unassigned variable.

- `undefined` is a global value that represents the absence of an assigned value.
  - `typeof undefined === 'undefined'`
- `null` is an object that indicates that a variable has been explicitly assigned "no value".
  - `typeof null === 'object'`

Setting a variable to `undefined` means the variable effectively does not exist. Some processes, such as JSON serialization, may strip `undefined` properties from objects. In contrast, `null` properties indicate will be preserved so you can explicitly convey the concept of an "empty" property.

The following evaluate to `undefined`:

- A variable when it is declared but not assigned a value (i.e. defined)
  - ```
    let foo;
    console.log('is undefined?', foo === undefined);
    // is undefined? true
    ```
- Accessing the value of a property that doesn't exist
  - ```
    let foo = { a: 'a' };
    console.log('is undefined?', foo.b === undefined);
    // is undefined? true
    ```
- The return value of a function that doesn't return a value
  - ```
    function foo() { return; }
    console.log('is undefined?', foo() === undefined);
    // is undefined? true
    ```
- The value of a function argument that is declared but has been omitted from the function call
  - ```
    function foo(param) {
      console.log('is undefined?', param === undefined);
    }
    foo('a');
    foo();
    // is undefined? false
    // is undefined? true
    ```

undefined is also a property of the global window object.

```
// Only in browsers
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

Version < 5

Before ECMAScript 5 you could actually change the value of the window.undefined property to any other value potentially breaking everything.

Section 3.5: Infinity and -Infinity

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

Infinity is a property of the global object (therefore a global variable) that represents mathematical infinity. It is a reference to Number.POSITIVE_INFINITY

It is greater than any other value, and you can get it by dividing by 0 or by evaluating the expression of a number that's so big that overflows. This actually means there is no division by 0 errors in JavaScript, there is Infinity!

There is also -Infinity which is mathematical negative infinity, and it's lower than any other value.

To get -Infinity you negate Infinity, or get a reference to it in Number.NEGATIVE_INFINITY.

```
- (Infinity); // -Infinity
```

Now let's have some fun with examples:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0 , yes there is a negative 0 in the language
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// Try your own!
```

Section 3.6: Number constants

The Number constructor has some built in constants that can be useful

```
Number.MAX_VALUE;          // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER;   // 9007199254740991
```

```
Number.MIN_VALUE;          // 5e-324
Number.MIN_SAFE_INTEGER;   // -9007199254740991

Number.EPSILON;            // 0.0000000000000002220446049250313

Number.POSITIVE_INFINITY;  // Infinity
Number.NEGATIVE_INFINITY;  // -Infinity

Number.NaN;                // NaN
```

In many cases the various operators in JavaScript will break with values outside the range of (`Number.MIN_SAFE_INTEGER`, `Number.MAX_SAFE_INTEGER`)

Note that `Number.EPSILON` represents the different between one and the smallest `Number` greater than one, and thus the smallest possible difference between two different `Number` values. One reason to use this is due to the nature of how numbers are stored by JavaScript see Check the equality of two numbers

Section 3.7: Operations that return NaN

Mathematical operations on values other than numbers return NaN.

```
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

An exception: Single-number arrays.

```
[2] * [3]  // Returns 6
```

Also, remember that the `+` operator concatenates strings.

```
"a" + "b"  // Returns "ab"
```

Dividing zero by zero returns `NaN`.

```
0 / 0         // NaN
```

Note: In mathematics generally (unlike in JavaScript programming), dividing by zero is not possible.

Section 3.8: Math library functions that return NaN

Generally, `Math` functions that are given non-numeric arguments will return NaN.

```
Math.floor("a")
```

The square root of a negative number returns NaN, because `Math.sqrt` does not support imaginary or complex numbers.

```
Math.sqrt(-1)
```

Chapter 4: Comments

Section 4.1: Using Comments

To add annotations, hints, or exclude some code from being executed JavaScript provides two ways of commenting code lines

Single line Comment `//`

Everything after the `//` until the end of the line is excluded from execution.

```
function elementAt( event ) {
  // Gets the element from Event coordinates
  return document.elementFromPoint(event.clientX, event.clientY);
}
// TODO: write more cool stuff!
```

Multi-line Comment `/**/`

Everything between the opening `/*` and the closing `*/` is excluded from execution, even if the opening and closing are on different lines.

```
/*
   Gets the element from Event coordinates.
   Use like:
   var clickedEl = someEl.addEventListener("click", elementAt, false);
*/
function elementAt( event ) {
  return document.elementFromPoint(event.clientX, event.clientY);
}
/* TODO: write more useful comments! */
```

Section 4.2: Using HTML comments in JavaScript (Bad practice)

HTML comments (optionally preceded by whitespace) will cause code (on the same line) to be ignored by the browser also, though this is considered **bad practice**.

One-line comments with the HTML comment opening sequence (`<!--`):

> **Note:** the JavaScript interpreter ignores the closing characters of HTML comments (`-->`) here.

```
<!-- A single-line comment.
<!-- --> Identical to using `//` since
<!-- --> the closing `-->` is ignored.
```

This technique can be observed in legacy code to hide JavaScript from browsers that didn't support it:

```
<script type="text/javascript" language="JavaScript">
<!--
/* Arbitrary JavaScript code.
   Old browsers would treat
   it as HTML code. */
// -->
```

```
</script>
```

An HTML closing comment can also be used in JavaScript (independent of an opening comment) at the beginning of a line (optionally preceded by whitespace) in which case it too causes the rest of the line to be ignored:

```
--> Unreachable JS code
```

These facts have also been exploited to allow a page to call itself first as HTML and secondly as JavaScript. For example:

```
<!--
self.postMessage('reached JS "file"');
/*
-->
<!DOCTYPE html>
<script>
var w1 = new Worker('#1');
w1.onmessage = function (e) {
    console.log(e.data); // 'reached JS "file"
};
</script>
<!--
*/
-->
```

When run a HTML, all the multiline text between the `<!--` and `-->` comments are ignored, so the JavaScript contained therein is ignored when run as HTML.

As JavaScript, however, while the lines beginning with `<!--` and `-->` are ignored, their effect is not to escape over *multiple* lines, so the lines following them (e.g., `self.postMessage(...`) will not be ignored when run as JavaScript, at least until they reach a *JavaScript* comment, marked by `/*` and `*/`. Such JavaScript comments are used in the above example to ignore the remaining *HTML* text (until the `-->` which is also ignored as JavaScript).

Chapter 5: Console

The information displayed by a debugging/web console is made available through the multiple methods of the `console` Javascript object that can be consulted through `console.dir(console)`. Besides the `console.memory` property, the methods displayed are generally the following (taken from Chromium's output):

- assert
- clear
- count
- debug
- dir
- dirxml
- error
- group
- groupCollapsed
- groupEnd
- info
- log
- markTimeline
- profile
- profileEnd
- table
- time
- timeEnd
- timeStamp
- timeline
- timelineEnd
- trace
- warn

Opening the Console

In most current browsers, the JavaScript Console has been integrated as a tab within Developer Tools. The shortcut keys listed below will open Developer Tools, it might be necessary to switch to the right tab after that.

Chrome

Opening the "Console" panel of Chrome's **DevTools**:

- Windows / Linux: any of the following options.
  - `Ctrl` + `Shift` + `J`
  - `Ctrl` + `Shift` + `I`, then click on the "Web Console" tab **or** press `ESC` to toggle the console on and off
  - `F12`, then click on the "Console" tab **or** press `ESC` to toggle the console on and off
- Mac OS: `Cmd` + `Opt` + `J`

Firefox

Opening the "Console" panel in Firefox's **Developer Tools**:

- Windows / Linux: any of the following options.
  - `Ctrl` + `Shift` + `K`
  - `Ctrl` + `Shift` + `I`, then click on the "Web Console" tab **or** press `ESC` to toggle the console on and off
  - `F12`, then click on the "Web Console" tab **or** press `ESC` to toggle the console on and off
- Mac OS: `Cmd` + `Opt` + `K`

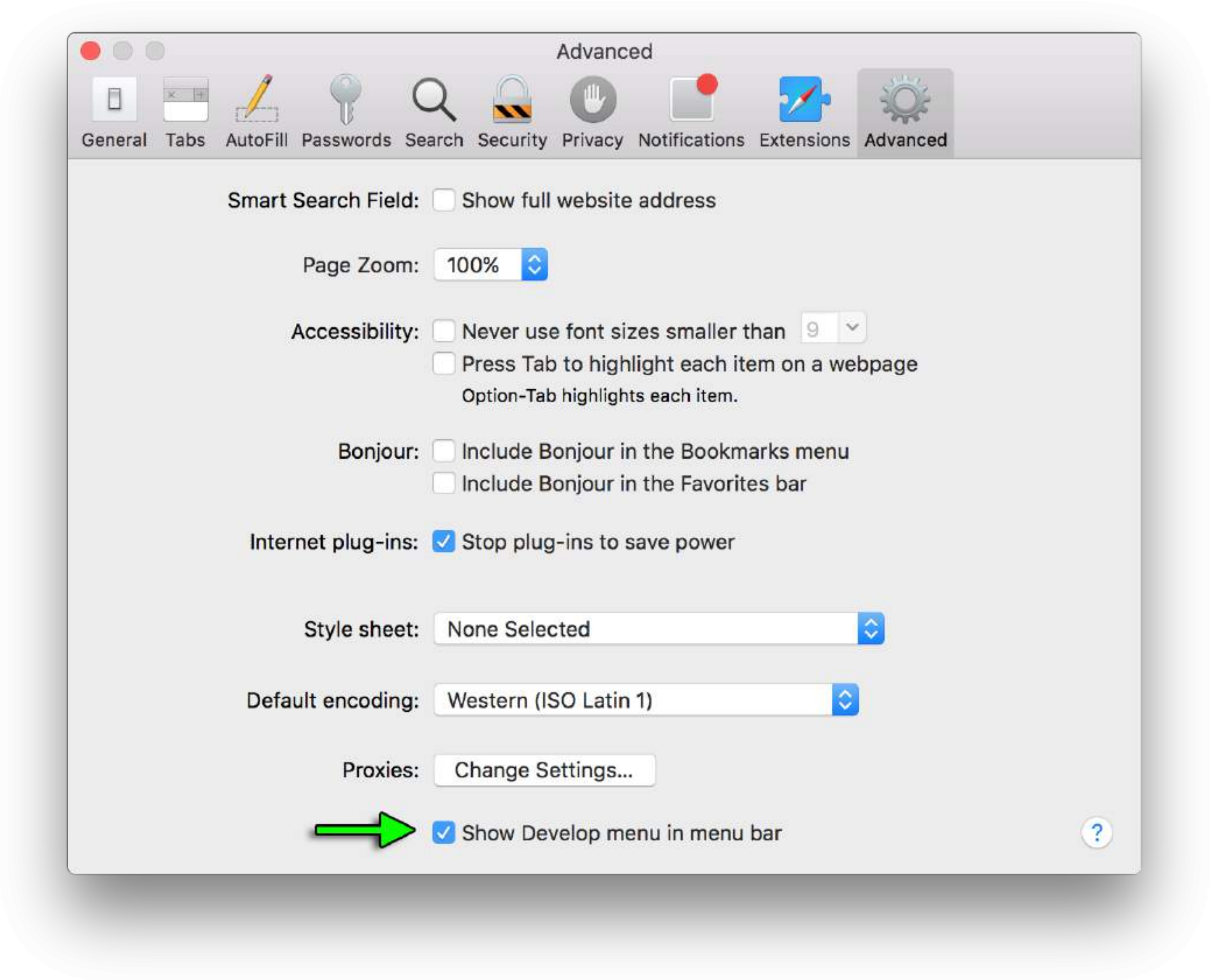
Edge and Internet Explorer

Opening the "Console" panel in the **F12 Developer Tools**:

- `F12`, then click on the "Console" tab

Safari

Opening the "Console" panel in Safari's **Web Inspector** you must first enable the develop menu in Safari's Preferences



Then you can either pick "Develop->Show Error Console" from the menus or press `⌘` + `Option` + `C`

Opera

Opening the “Console” in opera:

- `Ctrl` + `Shift` + `I`,then click on the “Console” tab

Compatibility

When using or emulating Internet Explorer 8 or earlier versions (e.g. through Compatibility View / Enterprise Mode) the console will **only** be defined when the Developer Tools are active, so `console.log()` statements can cause an exception and prevent code from executing. To mitigate this, you can check to see if the console is available before you log:

```
if (typeof window.console !== 'undefined')
{
   console.log("Hello World");
}
```

Or at the start of your script you can identify if the console is available and if not, define a null function to catch all of your references and prevent exceptions.

```
if (!window.console)
{
    console = {log: function() {}};
}
```

Note this second example will stop **all** console logs even if the developer window has been opened.

Using this second example will preclude use of other functions such as `console.dir(obj)` unless that is specifically added.

A browser's debugging console or [web console](https://developer.mozilla.org/en-US/docs/Tools/Web_Console) is generally used by developers to identify errors, understand flow of execution, log data and for many other purpose at runtime. This information is accessed through the [`console`](https://developer.mozilla.org/en-US/docs/Web/API/Console) object.

Section 5.1: Measuring time - console.time()

`console.time()` can be used to measure how long a task in your code takes to run.

Calling `console.time([label])` starts a new timer. When `console.timeEnd([label])` is called, the elapsed time, in milliseconds, since the original `.time()` call is calculated and logged. Because of this behavior, you can call `.timeEnd()` multiple times with the same label to log the elapsed time since the original `.time()` call was made.

Example 1:

```
console.time('response in');

alert('Click to continue');
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

will output:

```
response in: 774.967ms
response in: 1402.199ms
```

Example 2:

```
var elms = document.getElementsByTagName('*'); //select all elements on the page

console.time('Loop time');

for (var i = 0; i < 5000; i++) {
    for (var j = 0, length = elms.length; j < length; j++) {
        // nothing to do ...
    }
}

console.timeEnd('Loop time');
```

will output:

```
Loop time: 40.716ms
```

Section 5.2: Formatting console output

Many of the console's print methods can also handle C-like string formatting, using `%` tokens:

```
console.log('%s has %d points', 'Sam', 100);
```

Displays `Sam has 100 points`.

The full list of format specifiers in JavaScript is:

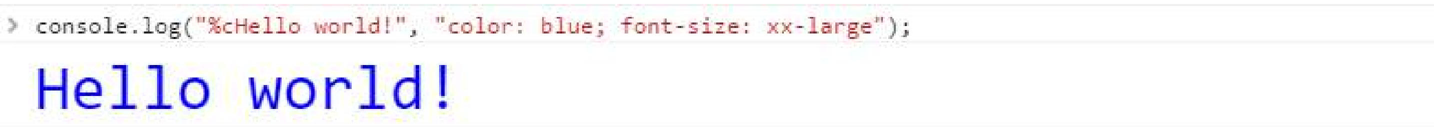
Specifier	Output
`%s`	Formats the value as a string
`%i` or `%d`	Formats the value as an integer
`%f`	Formats the value as a floating point value
`%o`	Formats the value as an expandable DOM element
`%O`	Formats the value as an expandable JavaScript object
`%c`	Applies CSS style rules to the output string as specified by the second parameter

Advanced styling

When the CSS format specifier (`%c`) is placed at the left side of the string, the print method will accept a second parameter with CSS rules which allow fine-grained control over the formatting of that string:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```

Displays:

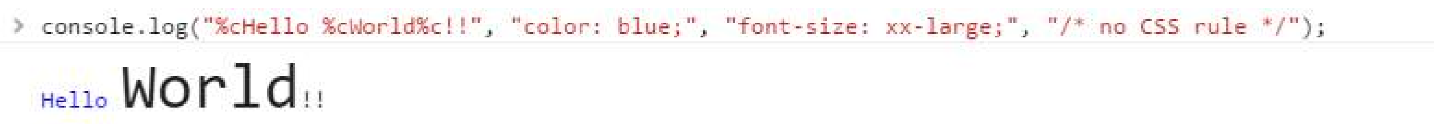


It is possible to use multiple `%c` format specifiers:

- any substring to the right of a `%c` has a corresponding parameter in the print method;
- this parameter may be an empty string, if there is no need to apply CSS rules to that same substring;
- if two `%c` format specifiers are found, the 1st (encased in `%c`) and 2nd substring will have their rules defined in the 2nd and 3rd parameter of the print method respectively.
- if three `%c` format specifiers are found, then the 1st, 2nd and 3rd substrings will have their rules defined in the 2nd , 3rd and 4th parameter respectively, and so on...

```
console.log("%cHello %cWorld%c!!", // string to be printed
            "color: blue;", // applies color formatting to the 1st substring
            "font-size: xx-large;", // applies font formatting to the 2nd substring
            "/* no CSS rule*/" // does not apply any rule to the remaining substring
);
```

Displays:



Using groups to indent output

Output can be indented and enclosed in a collapsible group in the debugging console with the following methods:

- `console.groupCollapsed()`: creates a collapsed group of entries that can be expanded through the disclosure button in order to reveal all the entries performed after this method is invoked;
- `console.group()`: creates an expanded group of entries that can be collapsed in order to hide the entries after this method is invoked.

The indentation can be removed for posterior entries by using the following method:

- console.groupEnd(): exits the current group, allowing newer entries to be printed in the parent group after this method is invoked.

Groups can be cascaded to allow multiple indented output or collapsible layers within each other:

`= Collapsed group expanded =>`

Section 5.3: Printing to a browser's debugging console

A browser's debugging console can be used in order to print simple messages. This debugging or web console can be directly opened in the browser (F12 key in most browsers – see *Remarks* below for further information) and the `log` method of the `console` JavaScript object can be invoked by typing the following:

```
console.log('My message');
```

Then, by pressing Enter, this will display `My message` in the debugging console.

`console.log()` can be called with any number of arguments and variables available in the current scope. Multiple arguments will be printed in one line with a small space between them.

```
var obj = { test: 1 };
console.log(['string'], 1, obj, window);
```

The `log` method will display the following in the debugging console:

```
['string']  1  Object { test: 1 }  Window { /* truncated */ }
```

Beside plain strings, `console.log()` can handle other types, like arrays, objects, dates, functions, etc.:

```
console.log([0, 3, 32, 'a string']);
console.log({ key1: 'value', key2: 'another value'});
```

Displays:

```
Array [0, 3, 32, 'a string']
Object { key1: 'value', key2: 'another value'}
```

Nested objects may be collapsed:

```
console.log({ key1: 'val', key2: ['one', 'two'], key3: { a: 1, b: 2 } });
```

Displays:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Certain types such as `Date` objects and **`function`**s may be displayed differently:

```
console.log(new Date(0));
console.log(function test(a, b) { return c; });
```

Displays:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)
function test(a, b) { return c; }
```

Other print methods

In addition to the `log` method, modern browsers also support similar methods:

- `console.info` – small informative icon (ⓘ) appears on the left side of the printed string(s) or object(s).
- `console.warn` – small warning icon (!) appears on the left side. In some browsers, the background of the log is yellow.
- `console.error` – small times icon (⊗) appears on the left side. In some browsers, the background of the log is red.
- `console.timeStamp` – outputs the current time and a specified string, but is non-standard:

  ```
  console.timeStamp('msg');
  ```

  Displays:

  ```
  00:00:00.001 msg
  ```

- `console.trace` – outputs the current stack trace or displays the same output as the `log` method if invoked in the global scope.

  ```
  function sec() {
     first();
  ```

```
}
function first() {
    console.trace();
}
sec();
```

Displays:

```
first
sec
(anonymous function)
```

The above image shows all the functions, with the exception of `timeStamp`, in Chrome version 56.

These methods behave similarly to the `log` method and in different debugging consoles may render in different colors or formats.

In certain debuggers, the individual objects information can be further expanded by clicking the printed text or a small triangle (►) which refers to the respective object properties. These collapsing object properties can be open or closed on log. See the `console.dir` for additional information on this

Section 5.4: Including a stack trace when logging - console.trace()

```
function foo() {
  console.trace('My log statement');
}

foo();
```

Will display this in the console:

```
My log statement       VM696:1
  foo                  @ VM696:1
  (anonymous function) @ (program):1
```

Note: Where available it's also useful to know that the same stack trace is accessible as a property of the Error object. This can be useful for post-processing and gathering automated feedback.

```
var e = new Error('foo');
console.log(e.stack);
```

Section 5.5: Tabulating values - console.table()

In most environments, `console.table()` can be used to display objects and arrays in a tabular format.

For example:

```
console.table(['Hello', 'world']);
```

displays like:

(index)	value
0	"Hello"
1	"world"

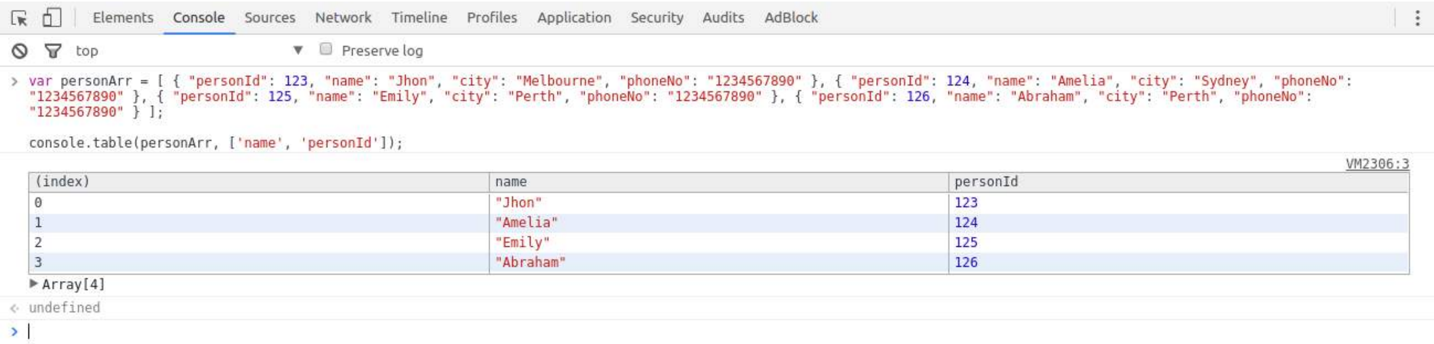
```
console.table({foo: 'bar', bar: 'baz'});
```

displays like:

(index)	value
"foo"	"bar"
"bar"	"baz"

```
var personArr = [
{
        "personId": 123,
        "name": "Jhon",
        "city": "Melbourne",
        "phoneNo": "1234567890"
},
{
        "personId": 124,
        "name": "Amelia",
        "city": "Sydney",
        "phoneNo": "1234567890"
},
{
        "personId": 125,
        "name": "Emily",
        "city": "Perth",
        "phoneNo": "1234567890"
},
{
        "personId": 126,
        "name": "Abraham",
        "city": "Perth",
        "phoneNo": "1234567890"
}
];
console.table(personArr, ['name', 'personId']);
```

displays like:



Section 5.6: Counting - console.count()

`console.count([obj])` places a counter on the object's value provided as argument. Each time this method is invoked, the counter is increased (with the exception of the empty string `''`). A label together with a number is displayed in the debugging console according to the following format:

```
[label]: X
```

`label` represents the value of the object passed as argument and `X` represents the counter's value.

An object's value is always considered, even if variables are provided as arguments:

```
var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
console.count('2');
console.count('');
```

Displays:

```
1: 1
2: 1
: 1
1: 2
2: 2
: 1
```

Strings with numbers are converted to `Number` objects:

```
console.count(42.3);
```

```
console.count(Number('42.3'));
console.count('42.3');
```

Displays:

```
42.3: 1
42.3: 2
42.3: 3
```

Functions point always to the global `Function` object:

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){};
console.count(fn1);
console.count(Number);
```

Displays:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Certain objects get specific counters associated to the type of object they refer to:

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
console.count(1/0);
console.count(String(1/0));
console.count(window);
console.count(document);
console.count(console);
console.count(console.__proto__);
console.count(console.constructor.prototype);
console.count(console.__proto__.constructor.prototype);
console.count(Object.getPrototypeOf(console));
console.count(null);
```

Displays:

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
```

```
[object Object]: 2
[object Object]: 3
[object Object]: 4
[object Object]: 5
null: 1
```

Empty string or absence of argument

If no argument is provided while **sequentially inputting the count method in the debugging console**, an empty string is assumed as parameter, i.e.:

```
> console.count();
  : 1
> console.count('');
  : 2
> console.count("");
  : 3
```

Section 5.7: Clearing the console - console.clear()

You can clear the console window using the `console.clear()` method. This removes all previously printed messages in the console and may print a message like "Console was cleared" in some environments.

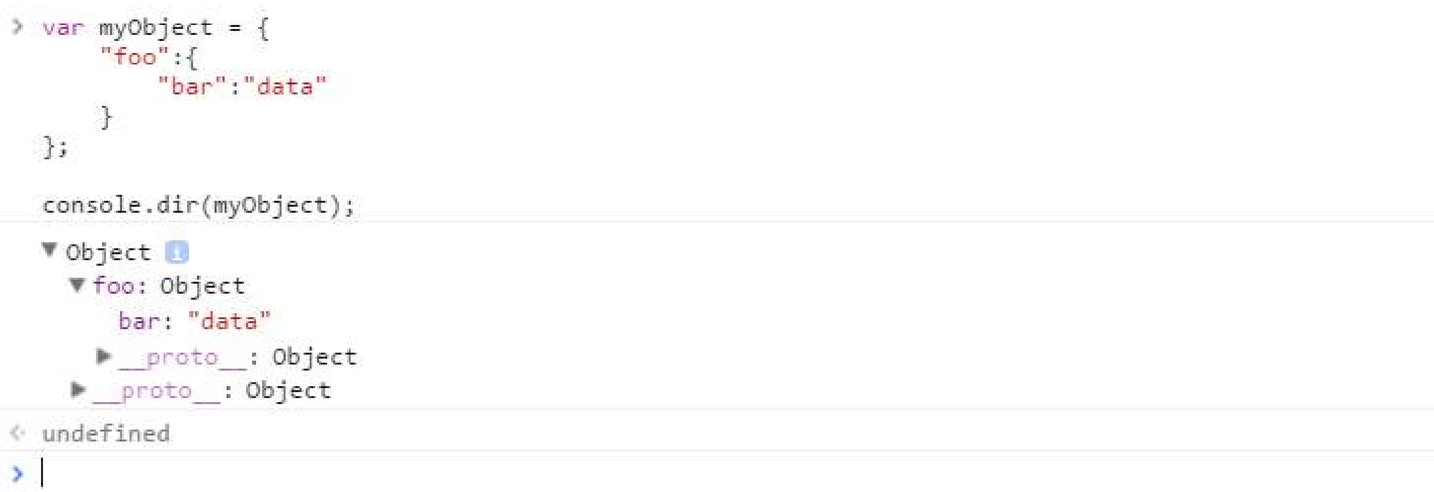
Section 5.8: Displaying objects and XML interactively - console.dir(), console.dirxml()

`console.dir(object)` displays an interactive list of the properties of the specified JavaScript object. The output is presented as a hierarchical listing with disclosure triangles that let you see the contents of child objects.

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dir(myObject);
```

displays:



`console.dirxml(object)` prints an XML representation of the descendant elements of object if possible, or the JavaScript representation if not. Calling `console.dirxml()` on HTML and XML elements is equivalent to calling `console.log()`.

Example 1:

```
console.dirxml(document)
```

displays:

Example 2:

```
console.log(document)
```

displays:

Example 3:

```
var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dirxml(myObject);
```

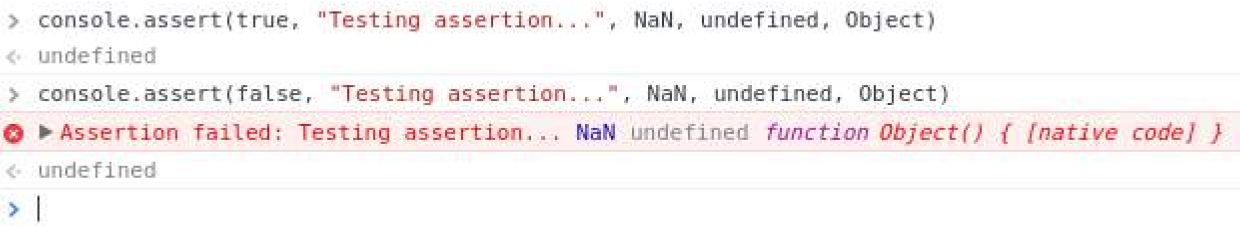
displays:

Section 5.9: Debugging with assertions - console.assert()

Writes an error message to the console if the assertion is **`false`**. Otherwise, if the assertion is **`true`**, this does nothing.

```
console.assert('one' === 1);
```

Multiple arguments can be provided after the assertion–these can be strings or other objects–that will only be printed if the assertion is **`false`**:



[`console.assert`] does *not* throw an `AssertionError` (except in Node.js), meaning that this method is incompatible with most testing frameworks and that code execution will not break on a failed assertion.

Chapter 6: Datatypes in JavaScript

Section 6.1: typeof

`typeof` is the 'official' function that one uses to get the `type` in JavaScript, however in certain cases it might yield some unexpected results ...

1. Strings

`typeof "String"` or
`typeof Date(2011,01,01)`

> "string"

2. Numbers

`typeof 42`

> "number"

3. Bool

`typeof true` (valid values `true` and `false`)

> "boolean"

4. Object

`typeof {}` or
`typeof []` or
`typeof null` or
`typeof /aaa/` or
`typeof Error()`

> "object"

5. Function

`typeof function(){}`

> "function"

6. Undefined

`var var1; typeof var1`

> "undefined"

Section 6.2: Finding an object's class

To find whether an object was constructed by a certain constructor or one inheriting from it, you can use the **instanceof** command:

```
//We want this function to take the sum of the numbers passed to it
//It can be called as sum(1, 2, 3) or sum([1, 2, 3]) and should give 6
function sum(...arguments) {
    if (arguments.length === 1) {
        const [firstArg] = arguments
        if (firstArg instanceof Array) { //firstArg is something like [1, 2, 3]
            return sum(...firstArg) //calls sum(1, 2, 3)
        }
    }
    return arguments.reduce((a, b) => a + b)
}

console.log(sum(1, 2, 3))   //6
console.log(sum([1, 2, 3])) //6
console.log(sum(4))         //4
```

Note that primitive values are not considered instances of any class:

```
console.log(2 instanceof Number)        //false
console.log('abc' instanceof String)    //false
console.log(true instanceof Boolean)    //false
console.log(Symbol() instanceof Symbol) //false
```

Every value in JavaScript besides **null** and **undefined** also has a `constructor` property storing the function that was used to construct it. This even works with primitives.

```
//Whereas instanceof also catches instances of subclasses,
//using obj.constructor does not
console.log([] instanceof Object, [] instanceof Array)           //true true
console.log([].constructor === Object, [].constructor === Array) //false true

function isNumber(value) {
    //null.constructor and undefined.constructor throw an error when accessed
    if (value === null || value === undefined) return false
    return value.constructor === Number
}
console.log(isNumber(null), isNumber(undefined))                  //false false
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1))     //false false false
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

Section 6.3: Getting object type by constructor name

When one with **typeof** operator one gets type `object` it falls into somewhat wast category...

In practice you might need to narrow it down to what sort of 'object' it actually is and one way to do it is to use object constructor name to get what flavour of object it actually is: `Object.prototype.toString.call(yourObject)`

1. String

```
Object.prototype.toString.call("String")
```

> "[object String]"

2. Number

`Object.prototype.toString.call(42)`

> "[object Number]"

3. Bool

`Object.prototype.toString.call(true)`

> "[object Boolean]"

4. Object

`Object.prototype.toString.call(Object())` or
`Object.prototype.toString.call({})`

> "[object Object]"

5. Function

`Object.prototype.toString.call(function(){})`

> "[object Function]"

6. Date

`Object.prototype.toString.call(new Date(2015,10,21))`

> "[object Date]"

7. Regex

`Object.prototype.toString.call(new RegExp())` or
`Object.prototype.toString.call(/foo/);`

> "[object RegExp]"

8. Array

`Object.prototype.toString.call([]);`

> "[object Array]"

9. Null

`Object.prototype.toString.call(null);`

> "[object Null]"

10. Undefined

```
Object.prototype.toString.call(undefined);
```

> "[object Undefined]"

11. Error

```
Object.prototype.toString.call(Error());
```

> "[object Error]"

Chapter 7: Strings

Section 7.1: Basic Info and String Concatenation

Strings in JavaScript can be enclosed in Single quotes `'hello'`, Double quotes `"Hello"` and (from ES2015, ES6) in Template Literals (*backticks*) `` `hello` ``.

```
var hello  = "Hello";
var world  = 'world';
var helloW = `Hello World`;              // ES2015 / ES6
```

Strings can be created from other types using the `String()` function.

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Or, `toString()` can be used to convert Numbers, Booleans or Objects to Strings.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Strings also can be created by using `String.fromCharCode` method.

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

Creating a String object using **new** keyword is allowed, but is not recommended as it behaves like Objects unlike primitive strings.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
typeof objectString.valueOf();//"string"
```

Concatenating Strings

String concatenation can be done with the **+** concatenation operator, or with the built-in `concat()` method on the String object prototype.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar);        // => "FooBar"
console.log(foo + " " + bar);  // => "Foo Bar"

foo.concat(bar)            // => "FooBar"
"a".concat("b", " ", "d")  // => "ab d"
```

Strings can be concatenated with non-string variables but will type-convert the non-string variables into strings.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

String Templates

Version ≥ 6

Strings can be created using template literals (*backticks*) `` `hello` ``.

```
var greeting = `Hello`;
```

With template literals, you can do string interpolation using `${variable}` inside template literals:

```
var place = `World`;
var greet = `Hello ${place}!`

console.log(greet); // "Hello World!"
```

You can use String.raw to get backslashes to be in the string without modification.

```
`a\\b` // =  a\b
String.raw`a\\b` // = a\\b
```

Section 7.2: Reverse String

The most "popular" way of reversing a string in JavaScript is the following code fragment, which is quite common:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}

reverseString('string');    // "gnirts"
```

However, this will work only so long as the string being reversed does not contain surrogate pairs. Astral symbols, i.e. characters outside of the basic multilingual plane, may be represented by two code units, and will lead this naive technique to produce wrong results. Moreover, characters with combining marks (e.g. diaeresis) will appear on the logical "next" character instead of the original one it was combined with.

```
'?????.'.split('').reverse().join(''); //fails
```

While the method will work fine for most languages, a truly accurate, encoding respecting algorithm for string reversal is slightly more involved. One such implementation is a tiny library called Esrever, which uses regular expressions for matching combining marks and surrogate pairs in order to perform the reversing perfectly.

Explanation

Section	Explanation	Result
`str`	The input string	`"string"`
`String.prototype.split( deliminator )`	Splits string `str` into an array. The parameter `""` means to split between each character.	`["s","t","r","i","n","g"]`
`Array.prototype.reverse()`	Returns the array from the split string with its elements in reverse order.	`["g","n","i","r","t","s"]`
`Array.prototype.join( deliminator )`	Joins the elements in the array together into a string. The `""` parameter means an empty deliminator (i.e., the elements of the array are put right next to each other).	`"gnirts"`

Using spread operator

Version ≥ 6

```
function reverseString(str) {
    return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow'));  // "wolfrevokcats"
console.log(reverseString(1337));             // "7331"
console.log(reverseString([1, 2, 3]));        // "3,2,1"
```

Custom `reverse()` function

```
function reverse(string) {
    var strRev = "";
    for (var i = string.length - 1; i >= 0; i--) {
        strRev += string[i];
    }
    return strRev;
}

reverse("zebra");  // "arbez"
```

Section 7.3: Comparing Strings Lexicographically

To compare strings alphabetically, use [`localeCompare()`](). This returns a negative value if the reference string is lexicographically (alphabetically) before the compared string (the parameter), a positive value if it comes afterwards, and a value of `0` if they are equal.

```
var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1
```

The `>` and `<` operators can also be used to compare strings lexicographically, but they cannot return a value of zero (this can be tested with the `==` equality operator). As a result, a form of the `localeCompare()` function can be written like so:

```
function strcmp(a, b) {
    if(a === b) {
        return 0;
    }

    if (a > b) {
        return 1;
    }

    return -1;
}

console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); //  0
console.log(strcmp("world", "hello")); //  1
```

This is especially useful when using a sorting function that compares based on the sign of the return value (such as `sort`).

```
var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
```

```
});
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

Section 7.4: Access character at index in string

Use charAt() to get a character at the specified index in the string.

```
var string = "Hello, World!";
console.log( string.charAt(4) ); // "o"
```

Alternatively, because strings can be treated like arrays, use the index via bracket notation.

```
var string = "Hello, World!";
console.log( string[4] ); // "o"
```

To get the character code of the character at a specified index, use charCodeAt().

```
var string = "Hello, World!";
console.log( string.charCodeAt(4) ); // 111
```

Note that these methods are all getter methods (return a value). Strings in JavaScript are immutable. In other words, none of them can be used to set a character at a position in the string.

Section 7.5: Escaping quotes

If your string is enclosed (i.e.) in single quotes you need to escape the inner literal quote with *backslash* \

```
var text = 'L\'albero means tree in Italian';
console.log( text ); \\ "L'albero means tree in Italian"
```

Same goes for double quotes:

```
var text = "I feel \"high\"";
```

Special attention must be given to escaping quotes if you're storing HTML representations within a String, since HTML strings make large use of quotations i.e. in attributes:

```
var content = "<p class=\"special\">Hello World!</p>";        // valid String
var hello   = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Quotes in HTML strings can also be represented using `'` (or `'`) as a single quote and `"` (or `"`) as double quotes.

```
var hi    = "<p class='special'>I'd like to say &quot;Hi&quot;</p>"; // valid String
var hello = '<p class="special">I&apos;d like to say "Hi"</p>';      // valid String
```

Note: The use of `'` and `"` will not overwrite double quotes that browsers can automatically place on attribute quotes. For example `<p class=special>` being made to `<p class="special">`, using `"` can lead to `<p class=""special"">` where `\"` will be `<p class="special">`.

Version ≥ 6

If a string has `'` and `"` you may want to consider using template literals (*also known as template strings in previous ES6 editions*), which do not require you to escape `'` and `"`. These use backticks (`` ` ``) instead of single or double quotes.

```
var x = `"Escaping " and ' can become very annoying`;
```

Section 7.6: Word Counter

Say you have a `<textarea>` and you want to retrieve info about the number of:

- Characters (total)
- Characters (no spaces)
- Words
- Lines

```
function wordCount( val ){
    var wom = val.match(/\S+/g);
    return {
        charactersNoSpaces : val.replace(/\s+/g, '').length,
        characters         : val.length,
        words              : wom ? wom.length : 0,
        lines              : val.split(/\r*\n/).length
    };
}


// Use like:
wordCount( someMultilineText ).words;   // (Number of words)
```

jsFiddle example

Section 7.7: Trim whitespace

To trim whitespace from the edges of a string, use `String.prototype.trim`:

```
"    some whitespaced string  ".trim();  // "some whitespaced string"
```

Many JavaScript engines, but not Internet Explorer, have implemented non-standard `trimLeft` and `trimRight` methods. There is a proposal, currently at Stage 1 of the process, for standardised `trimStart` and `trimEnd` methods, aliased to `trimLeft` and `trimRight` for compatibility.

```
// Stage 1 proposal
"    this is me    ".trimStart();  // "this is me    "
"    this is me    ".trimEnd();  // "    this is me"

// Non-standard methods, but currently implemented by most engines
"    this is me    ".trimLeft();  // "this is me    "
"    this is me    ".trimRight();  // "    this is me"
```

Section 7.8: Splitting a string into an array

Use `.split` to go from strings to an array of the split substrings:

```
var s = "one, two, three, four, five"
s.split(", ");  // ["one", "two", "three", "four", "five"]
```

Use the **array method** `.join` to go back to a string:

```
s.split(", ").join("--");  // "one--two--three--four--five"
```

Section 7.9: Strings are unicode

All JavaScript strings are unicode!

```
var s = "some ∆≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(5);  // 8710
```

There are no raw byte or binary strings in JavaScript. To effectively handle binary data, use Typed Arrays.

Section 7.10: Detecting a string

To detect whether a parameter is a *primitive* string, use **`typeof`**:

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string";   // true
typeof anInt === "string";     // false
typeof anObj === "string";     // false
```

If you ever have a `String` object, via `new String("somestr")`, then the above will not work. In this instance, we can use **`instanceof`**:

```
var aStringObj = new String("my string");
aStringObj instanceof String;    // true
```

To cover both instances, we can write a simple helper function:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

Or we can make use of `toString` function of `Object`. This can be useful if we have to check for other types as well say in a switch statement, as this method supports other datatypes as well just like **`typeof`**.

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString);//"[object String]"
Object.prototype.toString.call(oString);//"[object String]"
```

A more robust solution is to not *detect* a string at all, rather only check for what functionality is required. For example:

```
var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {

}
// Explicit check for the String substring prototype method
```

```
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}
```

Section 7.11: Substrings with slice

Use `.slice()` to extract substrings given two indices:

```
var s = "0123456789abcdefg";
s.slice(0, 5);  // "01234"
s.slice(5, 6);  // "5"
```

Given one index, it will take from that index to the end of the string:

```
s.slice(10);    // "abcdefg"
```

Section 7.12: Character code

The method `charCodeAt` retrieves the Unicode character code of a single character:

```
var charCode = "µ".charCodeAt(); // The character code of the letter µ is 181
```

To get the character code of a character in a string, the 0-based position of the character is passed as a parameter to `charCodeAt`:

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

Version ≥ 6

Some Unicode symbols don't fit in a single character, and instead require two UTF-16 surrogate pairs to encode. This is the case of character codes beyond 216 - 1 or 63553. These extended character codes or *code point* values can be retrieved with `codePointAt`:

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "????".codePointAt();
```

Section 7.13: String Representations of Numbers

JavaScript has native conversion from *Number* to its *String representation* for any base from *2 to 36*.

The most common representation after *decimal (base 10)* is *hexadecimal (base 16)*, but the contents of this section work for all bases in the range.

In order to convert a *Number* from decimal (base 10) to its hexadecimal (base 16) *String representation* the *toString* method can be used with *radix* `16`.

```
// base 10 Number
var b10 = 12;

// base 16 String representation
var b16 = b10.toString(16); // "c"
```

If the number represented is an integer, the inverse operation for this can be done with `parseInt` and the *radix* `16` again

```
// base 16 String representation
var b16 = 'c';

// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

To convert an arbitrary number (i.e. non-integer) from its *String representation* into a *Number*, the operation must be split into two parts; the integer part and the fraction part.

Version ≥ 6

```
let b16 = '3.243f3e0370cdc';
// Split into integer and fraction parts
let [i16, f16] = b16.split('.');

// Calculate base 10 integer part
let i10 = parseInt(i16, 16); // 3

// Calculate the base 10 fraction part
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988

// Put the base 10 parts together to find the Number
let b10 = i10 + f10; // 3.14159
```

Note 1: Be careful as small errors may be in the result due to differences in what is possible to be represented in different bases. It may be desirable to perform some kind of rounding afterwards.
Note 2: Very long representations of numbers may also result in errors due to the accuracy and maximum values of *Numbers* of the environment the conversions are happening in.

Section 7.14: String Find and Replace Functions

To search for a string inside a string, there are several functions:

`indexOf( searchString )` and `lastIndexOf( searchString )`

`indexOf()` will return the index of the first occurrence of `searchString` in the string. If `searchString` is not found, then `-1` is returned.

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

Similarly, `lastIndexOf()` will return the index of the last occurrence of `searchstring` or `-1` if not found.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") );   // 8
console.log( string.lastIndexOf("foo") ); // -1
```

`includes( searchString, start )`

`includes()` will return a boolean that tells whether `searchString` exists in the string, starting from index `start` (defaults to 0). This is better than `indexOf()` if you simply need to test for existence of a substring.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") );   // false
```

[replace(regexp|substring, replacement|replaceFunction)]

`replace()` will return a string that has all occurrences of substrings matching the [RegExp] `regexp` or string `substring` with a string `replacement` or the returned value of `replaceFunction`.

Note that this does not modify the string in place, but returns the string with replacements.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` can be used for conditional replacements for regular expression objects (i.e., with use with `regexp`). The parameters are in the following order:

Parameter	Meaning
`match`	the substring that matches the entire regular expressiong
`g1`, `g2`, `g3`, ...	the matching groups in the regular expression
`offset`	the offset of the match in the entire string
`string`	the entire string

Note that all parameters are optional.

```
var string = "heLlo, woRlD!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Section 7.15: Find the index of a substring inside a string

The `.indexOf` method returns the index of a substring inside another string (if exists, or -1 if otherwise)

```
'Hellow World'.indexOf('Wor');    // 7
```

`.indexOf` also accepts an additional numeric argument that indicates on what index should the function start looking

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

You should note that `.indexOf` is case sensitive

```
 'Hellow World'.indexOf('WOR');    // -1
```

Section 7.16: String to Upper Case

> String.prototype.toUpperCase():

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

Section 7.17: String to Lower Case

String.prototype.toLowerCase()

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Section 7.18: Repeat a String

Version ≥ 6

This can be done using the .repeat() method:

```
"abc".repeat(3);  // Returns "abcabcabc"
"abc".repeat(0);  // Returns ""
"abc".repeat(-1); // Throws a RangeError
```

Version < 6

In the general case, this should be done using a correct polyfill for the ES6 String.prototype.repeat() method. Otherwise, the idiom `new Array(n + 1).join(myString)` can repeat `n` times the string `myString`:

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString);  // Returns "abcabcabc"
```

Chapter 8: Date

Parameter	Details
`value`	The number of milliseconds since 1 January 1970 00:00:00.000 UTC (Unix epoch)
`dateAsString`	A date formatted as a string (see examples for more information)
`year`	The year value of the date. Note that `month` must also be provided, or the value will be interpreted as a number of milliseconds. Also note that values between `0` and `99` have special meaning. See the examples.
`month`	The month, in the range `0-11`. Note that using values outside the specified range for this and the following parameters will not result in an error, but rather cause the resulting date to "roll over" to the next value. See the examples.
`day`	Optional: The date, in the range `1-31`.
`hour`	Optional: The hour, in the range `0-23`.
`minute`	Optional: The minute, in the range `0-59`.
`second`	Optional: The second, in the range `0-59`.
`millisecond`	Optional: The millisecond, in the range `0-999`.

Section 8.1: Create a new Date object

To create a new `Date` object use the `Date()` constructor:

- **with no arguments**

  `Date()` creates a `Date` instance containing the current time (up to milliseconds) and date.

- **with one integer argument**

  `Date(m)` creates a `Date` instance containing the time and date corresponding to the Epoch time (1 January, 1970 UTC) plus `m` milliseconds. Example: `new Date(749019369738)` gives the date *Sun, 26 Sep 1993 04:56:09 GMT*.

- **with a string argument**

  `Date(dateString)` returns the `Date` object that results after parsing `dateString` with `Date.parse`.

- **with two or more integer arguments**

  `Date(i1, i2, i3, i4, i5, i6)` reads the arguments as year, month, day, hours, minutes, seconds, milliseconds and instantiates the corresponding `Date`object. Note that the month is 0-indexed in JavaScript, so 0 means January and 11 means December. Example: `new Date(2017, 5, 1)` gives *June 1st, 2017*.

Exploring dates

Note that these examples were generated on a browser in the Central Time Zone of the US, during Daylight Time, as evidenced by the code. Where comparison with UTC was instructive, `Date.prototype.toISOString()` was used to show the date and time in UTC (the Z in the formatted string denotes UTC).

```
// Creates a Date object with the current date and time from the
// user's browser
var now = new Date();
now.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)'
// true
// well, at the time of this writing, anyway

// Creates a Date object at the Unix Epoch (i.e., '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// Creates a Date object with the date and time 2,012 milliseconds
// after the Unix Epoch (i.e., '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// Creates a Date object with the first day of February of the year 2012
// in the local timezone.
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012 in the local
// timezone.
// (Months are zero-based)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012, in UTC.
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)'
// true
utc.toISOString() === '2012-01-01T00:00:00.000Z'
// true

// Parses a string into a Date object (ISO 8601 format added in ECMAScript 5.1)
// Implementations should assumed UTC because of ISO 8601 format and Z designation
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// Parses a string into a Date object (RFC in JavaScript 1.0)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Parses a string in no particular format, most of the time. Note that parsing
// logic in these cases is very implementation-dependent, and therefore can vary
// across browsers and versions.
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true, in Chrome 49 64-bit on Windows 10 in the en-US locale. Other versions in
// other locales may get a different result.

// Rolls values outside of a specified range to the next value.
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; note that the month rolled over to Feb; first the month rolled over to
// Jan based on the month 12 (11 being December), then again because of the day 32
// (January having 31 days).
```

```
// Special dates for years in the range 0-99
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)`
// true

// If you actually wanted to set the year to the year 12 CE, you'd need to use the
// setFullYear() method:
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01   12 00:00:00 GMT-0600 (Central Standard Time)`
// true
```

Section 8.2: Convert to a string format

Convert to String

```
var date1 = new Date();
date1.toString();
```

> Returns: "Fri Apr 15 2016 07:48:48 GMT-0400 (Eastern Daylight Time)"

Convert to Time String

```
var date1 = new Date();
date1.toTimeString();
```

> Returns: "07:48:48 GMT-0400 (Eastern Daylight Time)"

Convert to Date String

```
var date1 = new Date();
date1.toDateString();
```

> Returns: "Thu Apr 14 2016"

Convert to UTC String

```
var date1 = new Date();
date1.toUTCString();
```

> Returns: "Fri, 15 Apr 2016 11:48:48 GMT"

Convert to ISO String

```
var date1 = new Date();
date1.toISOString();
```

> Returns: "2016-04-14T23:49:08.596Z"

Convert to GMT String

```
var date1 = new Date();
date1.toGMTString();
```

> Returns: "Thu, 14 Apr 2016 23:49:08 GMT"

This function has been marked as deprecated so some browsers may not support it in the future. It is suggested to use toUTCString() instead.

Convert to Locale Date String

```
var date1 = new Date();
date1.toLocaleDateString();
```

> Returns: "4/14/2016"

This function returns a locale sensitive date string based upon the user's location by default.

```
date1.toLocaleDateString([locales [, options]])
```

can be used to provide specific locales but is browser implementation specific. For example,

```
date1.toLocaleDateString(["zh", "en-US"]);
```

would attempt to print the string in the Chinese locale using United States English as a fallback. The options parameter can be used to provide specific formatting. For example:

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

would result in

> "Thursday, April 14, 2016".

See the MDN for more details.

Section 8.3: Creating a Date from UTC

By default, a `Date` object is created as local time. This is not always desirable, for example when communicating a date between a server and a client that do not reside in the same timezone. In this scenario, one doesn't want to worry about timezones at all until the date needs to be displayed in local time, if that is even required at all.

The problem

In this problem we want to communicate a specific date (day, month, year) with someone in a different timezone. The first implementation naively uses local times, which results in wrong results. The second implementation uses UTC dates to avoid timezones where they are not needed.

Naive approach with WRONG results

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"];
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(2000,0,1);
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
      birthday.getMonth(), birthday.getFullYear()));

sendToBar(birthday.getTime());
```

> Sample output:
>
> `Foo was born on: Sat Jan 1 2000`

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
      birthday.getMonth(), birthday.getFullYear()));
```

> Sample output:
>
> `Foo was born on: Fri Dec 31 1999`

And thus, Bar would always believe Foo was born on the last day of 1999.

Correct approach

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"];
  var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(Date.UTC(2000,0,1));
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
      birthday.getUTCMonth(), birthday.getUTCFullYear()));

sendToBar(birthday.getTime());
```

> Sample output:
>
> `Foo was born on: Sat Jan 1 2000`

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
```

```
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
      birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

> Sample output:
>
> `Foo was born on: Sat Jan 1 2000`

Creating a Date from UTC

If one wants to create a `Date` object based on UTC or GMT, the `Date.UTC(...)` method can be used. It uses the same arguments as the longest `Date` constructor. This method will return a number representing the time that has passed since January 1, 1970, 00:00:00 UTC.

```
console.log(Date.UTC(2000,0,31,12));
```

> Sample output:
>
> `949320000000`

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
console.log(utcDate);
```

> Sample output:
>
> `Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))`

Unsurprisingly, the difference between UTC time and local time is, in fact, the timezone offset converted to milliseconds.

```
var utcDate = new Date(Date.UTC(2000,0,31,12));
var localDate = new Date(2000,0,31,12);

console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

> Sample output: `true`

Changing a Date object

All `Date` object modifiers, such as `setDate(...)` and `setFullYear(...)` have an equivalent takes an argument in UTC time rather than in local time.

```
var date = new Date();
date.setUTCFullYear(2000,0,31);
date.setUTCHours(12,0,0,0);
console.log(date);
```

> Sample output:

> Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))

The other UTC-specific modifiers are `.setUTCMonth()`, `.setUTCDate()` (for the day of the month), `.setUTCMinutes()`, `.setUTCSeconds()` and `.setUTCMilliseconds()`.

Avoiding ambiguity with getTime() and setTime()

Where the methods above are required to differentiate between ambiguity in dates, it is usually easier to communicate a date as the amount of time that has passed since January 1, 1970, 00:00:00 UTC. This single number represents a single point in time, and can be converted to local time whenever necessary.

```
var date = new Date(Date.UTC(2000,0,31,12));
var timestamp = date.getTime();
//Alternatively
var timestamp2 = Date.UTC(2000,0,31,12);
console.log(timestamp === timestamp2);
```

> Sample output: `true`

```
//And when constructing a date from it elsewhere...
var otherDate = new Date(timestamp);

//Represented as a universal date
console.log(otherDate.toUTCString());
//Represented as a local date
console.log(otherDate);
```

> Sample output:
>
> ```
> Mon, 31 Jan 2000 12:00:00 GMT
> Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))
> ```
>
> /code>

Section 8.4: Formatting a JavaScript date

Formatting a JavaScript date in modern browsers

In modern browsers (*), [`Date.prototype.toLocaleDateString()`] allows you to define the formatting of a `Date` in a convenient manner.

It requires the following format :

```
dateObj.toLocaleDateString([locales [, options]])
```

The `locales` parameter should be a string with a BCP 47 language tag, or an array of such strings.

The `options` parameter should be an object with some or all of the following properties:

- **localeMatcher** : possible values are `"lookup"` and `"best fit"`; the default is `"best fit"`
- **timeZone** : the only value implementations must recognize is `"UTC"`; the default is the runtime's default time

zone

- **hour12** :possible values are `true` and `false`; the default is locale dependent
- **formatMatcher** : possible values are `"basic"` and `"best fit"`; the default is `"best fit"`
- **weekday** : possible values are `"narrow"`, `"short"` & `"long"`
- **era** : possible values are `"narrow"`, `"short"` & `"long"`
- **year** : possible values are `"numeric"` & `"2-digit"`
- **month** : possible values are `"numeric"`, `"2-digit"`, `"narrow"`, `"short"` & `"long"`
- **day** : possible values are `"numeric"` & `"2-digit"`
- **hour** : possible values are `"numeric"` & `"2-digit"`
- **minute** : possible values are `"numeric"` & `"2-digit"`
- **second** : possible values are `"numeric"` & `"2-digit"`
- **timeZoneName** : possible values are `"short"` & `"long"`

How to use

```
var today = new Date().toLocaleDateString('en-GB', {
    day : 'numeric',
    month : 'short',
    year : 'numeric'
});
```

Output if executed on January 24th, 2036 :

```
'24 Jan 2036'
```

Going custom

If `Date.prototype.toLocaleDateString()` isn't flexible enough to fulfill whatever need you may have, you might want to consider creating a custom Date object that looks like this:

```
var DateObject = (function() {
    var monthNames = [
      "January", "February", "March",
      "April", "May", "June", "July",
      "August", "September", "October",
      "November", "December"
    ];
    var date = function(str) {
        this.set(str);
    };
    date.prototype = {
        set : function(str) {
            var dateDef = str ? new Date(str) : new Date();
            this.day = dateDef.getDate();
            this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
            this.month = dateDef.getMonth() + 1;
            this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
            this.monthName = monthNames[this.month - 1];
            this.year = dateDef.getFullYear();
        },
        get : function(properties, separator) {
            var separator = separator ? separator : '-'
                ret = [];
            for(var i in properties) {
                ret.push(this[properties[i]]);
            }
            return ret.join(separator);
```

```
        }
    };
    return date;
})();
```

If you included that code and executed `new DateObject()` on January 20th, 2019, it would produce an object with the following properties:

```
day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
year: 2019
```

To get a formatted string, you could do something like this:

```
new DateObject().get(['dayPadded', 'monthPadded', 'year']);
```

That would produce the following output:

```
20-01-2016
```

(*) **According to the MDN**, "modern browsers" means Chrome 24+, Firefox 29+, IE11, Edge12+, Opera 15+ & Safari **nightly build**

Section 8.5: Get the number of milliseconds elapsed since 1 January 1970 00:00:00 UTC

The static method `Date.now` returns the number of milliseconds that have elapsed since 1 January 1970 00:00:00 UTC. To get the number of milliseconds that have elapsed since that time using an instance of a `Date` object, use its `getTime` method.

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Section 8.6: Get the current time and date

Use `new Date()` to generate a new `Date` object containing the current date and time.

Note that `Date()` called without arguments is equivalent to `new Date(Date.now())`.

Once you have a date object, you can apply any of the several available methods to extract its properties (e.g. `getFullYear()` to get the 4-digits year).

Below are some common date methods.

Get the current year

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016
```

Get the current month

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

Please note that 0 = January. This is because months range from *0* to *11*, so it is often desirable to add `+1` to the index.

Get the current day

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Get the current hour

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Get the current minutes

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Get the current seconds

```
var seconds = (new Date()).getSeconds();
console.log(second);
// Sample output: 48
```

Get the current milliseconds

To get the milliseconds (ranging from 0 to 999) of an instance of a `Date` object, use its `getMilliseconds` method.

```
var milliseconds = (new Date()).getMilliseconds();
console.log(milliseconds);
 // Output: milliseconds right now
```

Convert the current time and date to a human-readable string

```
var now = new Date();
// convert date to a string in UTC timezone format:
console.log(now.toUTCString());
// Output: Wed, 21 Jun 2017 09:13:01 GMT
```

The static method `Date.now()` returns the number of milliseconds that have elapsed since 1 January 1970 00:00:00 UTC. To get the number of milliseconds that have elapsed since that time using an instance of a `Date` object, use its `getTime` method.

```
// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());
```

Section 8.7: Increment a Date Object

To increment date objects in JavaScript, we can usually do this:

```
var checkoutDate = new Date();    // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)

checkoutDate.setDate( checkoutDate.getDate() + 1 );
```

```
console.log(checkoutDate); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

It is possible to use `setDate` to change the date to a day in the following month by using a value larger than the number of days in the current month -

```
var checkoutDate = new Date();    // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 12 );
console.log(checkoutDate); // Tue Aug 02 2016 10:05:13 GMT-0400 (EDT)
```

The same applies to other methods such as getHours(), getMonth(),etc.

Adding Work Days

If you wish to add work days (in this case I am assuming Monday - Friday) you can use the `setDate` function although you need a little extra logic to account for the weekends (obviously this will not take account of national holidays) -

```
function addWorkDays(startDate, days) {
    // Get the day of the week as a number (0 = Sunday, 1 = Monday, .... 6 = Saturday)
    var dow = startDate.getDay();
    var daysToAdd = days;
    // If the current day is Sunday add one day
    if (dow == 0)
        daysToAdd++;
    // If the start date plus the additional days falls on or after the closest Saturday calculate
weekends
    if (dow + daysToAdd >= 6) {
        //Subtract days in current working week from work days
        var remainingWorkDays = daysToAdd - (5 - dow);
        //Add current working week's weekend
        daysToAdd += 2;
        if (remainingWorkDays > 5) {
            //Add two days for each working week by calculating how many weeks are included
            daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
            //Exclude final weekend if remainingWorkDays resolves to an exact number of weeks
            if (remainingWorkDays % 5 == 0)
                daysToAdd -= 2;
        }
    }
    startDate.setDate(startDate.getDate() + daysToAdd);
    return startDate;
}
```

Section 8.8: Convert to JSON

```
var date1 = new Date();
date1.toJSON();
```

> Returns: "2016-04-14T23:49:08.596Z"

Chapter 9: Date Comparison

Section 9.1: Comparing Date values

To check the equality of `Date` values:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

> Sample output: `false`

Note that you must use `valueOf()` or `getTime()` to compare the values of `Date` objects because the equality operator will compare if two object references are the same. For example:

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

> Sample output: `false`

Whereas if the variables point to the same object:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

> Sample output: `true`

However, the other comparison operators will work as usual and you can use `<` and `>` to compare that one date is earlier or later than the other. For example:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

> Sample output: `true`

It works even if the operator includes equality:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

> Sample output: `true`

Section 9.2: Date Difference Calculation

To compare the difference of two dates, we can do the comparison based on the timestamp.

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //convert milliseconds into years

console.log("Date difference in years : " + dateDiffInYears);
```

Chapter 10: Comparison Operations

Section 10.1: Abstract equality / inequality and type conversion

The Problem

The abstract equality and inequality operators (`==` and `!=`) convert their operands if the operand types do not match. This type coercion is a common source of confusion about the results of these operators, in particular, these operators aren't always transitive as one would expect.

```
""  ==  0;     // true A
 0  == "0";    // true A
""  == "0";    // false B
false == 0;    // true
false == "0";  // true

""  !=  0;     // false A
 0  != "0";    // false A
""  != "0";    // true B
false != 0;    // false
false != "0";  // false
```

The results start to make sense if you consider how JavaScript converts empty strings to numbers.

```
Number("");    // 0
Number("0");   // 0
Number(false); // 0
```

The Solution

In the statement `false B`, both the operands are strings (`""` and `"0"`), hence there will be **no type conversion** and since `""` and `"0"` are not the same value, `"" == "0"` is `false` as expected.

One way to eliminate unexpected behavior here is making sure that you always compare operands of the same type. For example, if you want the results of numerical comparison use explicit conversion:

```
var test = (a,b) => Number(a) == Number(b);
test("", 0);        // true;
test("0", 0);       // true
test("", "0");      // true;
test("abc", "abc"); // false as operands are not numbers
```

Or, if you want string comparison:

```
var test = (a,b) => String(a) == String(b);
test("", 0);   // false;
test("0", 0);  // true
test("", "0"); // false;
```

Side-note: `Number("0")` and `new Number("0")` isn't the same thing! While the former performs a type conversion, the latter will create a new object. Objects are compared by reference and not by value which explains the results below.

```
Number("0") == Number("0");         // true;
```

```
new Number("0") == new Number("0"); // false
```

Finally, you have the option to use strict equality and inequality operators which will not perform any implicit type conversions.

```
"" ===  0;  // false
 0 === "0"; // false
"" === "0"; // false
```

Further reference to this topic can be found here:

[Which equals operator (== vs ===) should be used in JavaScript comparisons?](#).

Abstract Equality (==)

Section 10.2: NaN Property of the Global Object

`NaN` ("**N**ot **a** **N**umber") is a special value defined by the *IEEE Standard for Floating-Point Arithmetic*, which is used when a non-numeric value is provided but a number is expected (`1 * "two"`), or when a calculation doesn't have a valid `number` result (`Math.sqrt(-1)`).

Any equality or relational comparisons with `NaN` returns `false`, even comparing it with itself. Because, `NaN` is supposed to denote the result of a nonsensical computation, and as such, it isn't equal to the result of any other nonsensical computations.

```
(1 * "two") === NaN  //false

NaN === 0;          // false
NaN === NaN;        // false
Number.NaN === NaN; // false

NaN < 0;            // false
NaN > 0;            // false
NaN > 0;            // false
NaN >= NaN;         // false
NaN >= 'two';       // false
```

Non-equal comparisons will always return `true`:

```
NaN !== 0;          // true
NaN !== NaN;        // true
```

Checking if a value is NaN

`Version ≥ 6`

You can test a value or expression for `NaN` by using the function Number.isNaN():

```
Number.isNaN(NaN);         // true
Number.isNaN(0 / 0);       // true
Number.isNaN('str' - 12);  // true

Number.isNaN(24);          // false
Number.isNaN('24');        // false
Number.isNaN(1 / 0);       // false
Number.isNaN(Infinity);    // false

Number.isNaN('str');       // false
Number.isNaN(undefined);   // false
```

```
Number.isNaN({});           // false
```

Version < 6

You can check if a value is **NaN** by comparing it with itself:

```
value !== value;    // true for NaN, false for any other value
```

You can use the following polyfill for `Number.isNaN()`:

```
Number.isNaN = Number.isNaN || function(value) {
    return value !== value;
}
```

By contrast, the global function `isNaN()` returns **true** not only for **NaN**, but also for any value or expression that cannot be coerced into a number:

```
isNaN(NaN);         // true
isNaN(0 / 0);       // true
isNaN('str' - 12);  // true

isNaN(24);          // false
isNaN('24');        // false
isNaN(Infinity);    // false

isNaN('str');       // true
isNaN(undefined);   // true
isNaN({});          // true
```

ECMAScript defines a “sameness” algorithm called `SameValue` which, since ECMAScript 6, can be invoked with `Object.is`. Unlike the `==` and `===` comparison, using `Object.is()` will treat **NaN** as identical with itself (and `-0` as not identical with `+0`):

```
Object.is(NaN, NaN)      // true
Object.is(+0, 0)         // false

NaN === NaN              // false
+0 === 0                 // true
```

Version < 6

You can use the following polyfill for `Object.is()` (from MDN):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 != -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

Points to note

NaN itself is a number, meaning that it does not equal to the string "NaN", and most importantly (though perhaps unintuitively):

```
typeof(NaN) === "number"; //true
```

Section 10.3: Short-circuiting in boolean operators

The and-operator (`&&`) and the or-operator (`||`) employ short-circuiting to prevent unnecessary work if the outcome of the operation does not change with the extra work.

In `x && y`, `y` will not be evaluated if `x` evaluates to **`false`**, because the whole expression is guaranteed to be **`false`**.

In `x || y`, `y` will not be evaluated if `x` evaluated to **`true`**, because the whole expression is guaranteed to be **`true`**.

Example with functions

Take the following two functions:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
  console.log("F");
  return false;
}
```

Example 1

```
T() && F(); // false
```

Output:

> 'T'
> 'F'

Example 2

```
F() && T(); // false
```

Output:

> 'F'

Example 3

```
T() || F(); // true
```

Output:

> 'T'

Example 4

```
F() || T(); // true
```

Output:

> 'F'
> 'T'

Short-circuiting to prevent errors

```
var obj; // object has value of undefined
if(obj.property){ }// TypeError: Cannot read property 'property' of undefined
if(obj.property && obj !== undefined){}// Line A TypeError: Cannot read property 'property' of
undefined
```

Line A: if you reverse the order the first conditional statement will prevent the error on the second by not executing it if it would throw the error

```
if(obj !== undefined && obj.property){}; // no error thrown
```

But should only be used if you expect `undefined`

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Short-circuiting to provide a default value

The `||` operator can be used to select either a "truthy" value, or the default value.

For example, this can be used to ensure that a nullable value is converted to a non-nullable value:

```
var nullableObj = null;
var obj = nullableObj || {};  // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

Or to return the first truthy value

```
var truthyValue = {x: 10};
return truthyValue || {}; // will return {x: 10}
```

The same can be used to fall back multiple times:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Short-circuiting to call an optional function

The `&&` operator can be used to evaluate a callback, only if it is passed:

```
function myMethod(cb) {
    // This can be simplified
    if (cb) {
       cb();
    }
```

```
    // To this
    cb && cb();
}
```

Of course, the test above does not validate that `cb` is in fact a `function` and not just an `Object`/`Array`/`String`/`Number`.

Section 10.4: Null and Undefined

The differences between `null` and `undefined`

`null` and `undefined` share abstract equality `==` but not strict equality `===`,

```
null == undefined   // true
null === undefined  // false
```

They represent slightly different things:

- `undefined` represents the *absence of a value*, such as before an identifier/Object property has been created or in the period between identifier/Function parameter creation and it's first set, if any.
- `null` represents the ***intentional*** *absence of a value* for an identifier or property which has already been created.

They are different types of syntax:

- `undefined` is a *property of the global Object*, usually immutable in the global scope. This means anywhere you can define an identifier other than in the global namespace could hide `undefined` from that scope (although things can still **be** `undefined`)
- `null` is a *word literal*, so it's meaning can never be changed and attempting to do so will throw an *Error*.

The similarities between `null` and `undefined`

`null` and `undefined` are both falsy.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Neither `null` or `undefined` equal `false` (see this question).

```
false == undefined   // false
false == null        // false
false === undefined  // false
false === null       // false
```

Using `undefined`

- If the current scope can't be trusted, use something which evaluates to *undefined*, for example `void 0;`.
- If `undefined` is shadowed by another value, it's just as bad as shadowing `Array` or `Number`.
- Avoid *setting* something as `undefined`. If you want to remove a property *bar* from an *Object* `foo`, `delete foo.bar;` instead.
- Existence testing identifier `foo` against `undefined` **could throw a Reference Error**, use `typeof foo` against `"undefined"` instead.

Section 10.5: Abstract Equality (==)

Operands of the abstract equality operator are compared *after* being converted to a common type. How this conversion happens is based on the specification of the operator:

[Specification for the == operator:](#)

> **7.2.13 Abstract Equality Comparison**
> The comparison `x == y`, where `x` and `y` are values, produces **`true`** or **`false`**. Such a comparison is performed as follows:
>
> 1. If `Type(x)` is the same as `Type(y)`, then:
>
> - **a.** Return the result of performing Strict Equality Comparison `x === y`.
>
> 2. If `x` is **`null`** and `y` is **`undefined`**, return **`true`**.
> 3. If `x` is **`undefined`** and `y` is **`null`**, return **`true`**.
> 4. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x == ToNumber(y)`.
> 5. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `ToNumber(x) == y`.
> 6. If `Type(x)` is `Boolean`, return the result of the comparison `ToNumber(x) == y`.
> 7. If `Type(y)` is `Boolean`, return the result of the `comparison x == ToNumber(y)`.
> 8. If `Type(x)` is either `String`, `Number`, or `Symbol` and `Type(y)` is `Object`, return the result of the comparison `x == ToPrimitive(y)`.
> 9. If `Type(x)` is Object and `Type(y)` is either `String`, `Number`, or `Symbol`, return the result of the comparison `ToPrimitive(x) == y`.
> 10. Return **`false`**.

Examples:

```
1 == 1;                     // true
1 == true;                  // true  (operand converted to number: true => 1)
1 == '1';                   // true  (operand converted to number: '1' => 1 )
1 == '1.00';                // true
1 == '1.000000000001';      // false
1 == '1.00000000000000001'; // true  (true due to precision loss)
null == undefined;          // true  (spec #2)
1 == 2;                     // false
0 == false;                 // true
0 == undefined;             // false
0 == "";                    // true
```

Section 10.6: Logic Operators with Booleans

```
var x = true,
    y = false;
```

AND

This operator will return true if both of the expressions evaluate to true. This boolean operator will employ short-circuiting and will not evaluate `y` if `x` evaluates to **`false`**.

```
x && y;
```

This will return false, because `y` is false.

OR

This operator will return true if one of the two expressions evaluate to true. This boolean operator will employ short-circuiting and y will not be evaluated if x evaluates to **true**.

```
x || y;
```

This will return true, because x is true.

NOT

This operator will return false if the expression on the right evaluates to true, and return true if the expression on the right evaluates to false.

```
!x;
```

This will return false, because x is true.

Section 10.7: Automatic Type Conversions

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

```
var x = "Hello";     // typeof x is a string
x = 5;               // changes typeof x to a number
```

When doing mathematical operations, JavaScript can convert numbers to strings:

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

```
"Hello" - "Dolly"    // returns NaN
```

Section 10.8: Logic Operators with Non-boolean values (boolean coercion)

Logical OR (||), reading left to right, will evaluate to the first *truthy* value. If no *truthy* value is found, the last value is returned.

```
var a = 'hello' || '';             // a = 'hello'
var b = '' || [];                  // b = []
var c = '' || undefined;           // c = undefined
var d = 1 || 5;                    // d = 1
var e = 0 || {};                   // e = {}
var f = 0 || '' || 5;              // f = 5
var g = '' || 'yay' || 'boo';      // g = 'yay'
```

Logical AND (&&), reading left to right, will evaluate to the first *falsy* value. If no *falsey* value is found, the last value is returned.

```
var a = 'hello' && '';                  // a = ''
var b = '' && [];                       // b = ''
var c = undefined && 0;                 // c = undefined
var d = 1 && 5;                         // d = 5
var e = 0 && {};                        // e = 0
var f = 'hi' && [] && 'done';           // f = 'done'
var g = 'bye' && undefined && 'adios';  // g = undefined
```

This trick can be used, for example, to set a default value to a function argument (prior to ES6).

```
var foo = function(val) {
    // if val evaluates to falsey, 'default' will be returned instead.
    return val || 'default';
}

console.log( foo('burger') );  // burger
console.log( foo(100) );       // 100
console.log( foo([]) );        // []
console.log( foo(0) );         // default
console.log( foo(undefined) ); // default
```

Just keep in mind that for arguments, `0` and (to a lesser extent) the empty string are also often valid values that should be able to be explicitly passed and override a default, which, with this pattern, they won't (because they are *falsy*).

Section 10.9: Empty Array

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */
[] == false; // true
```

When `[].toString()` is executed it calls `[].join()` if it exists, or `Object.prototype.toString()` otherwise. This comparison is returning **`true`** because `[].join()` returns `''` which, coerced into `0`, is equal to false [ToNumber](#).

Beware though, all objects are truthy and `Array` is an instance of `Object`:

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Section 10.10: Equality comparison operations

JavaScript has four different equality comparison operations.

SameValue

It returns **`true`** if both operands belong to the same Type and are the same value.

Note: the value of an object is a reference.

You can use this comparison algorithm via `Object.is` (ECMAScript 6).

Examples:

```
Object.is(1, 1);            // true
Object.is(+0, -0);          // false
Object.is(NaN, NaN);        // true
Object.is(true, "true");    // false
Object.is(false, 0);        // false
```

```
Object.is(null, undefined); // false
Object.is(1, "1");          // false
Object.is([], []);          // false
```

This algorithm has the properties of an equivalence relation:

- Reflexivity: `Object.is(x, x)` is **true**, for any value `x`
- Symmetry: `Object.is(x, y)` is **true** if, and only if, `Object.is(y, x)` is **true**, for any values `x` and `y`.
- Transitivity: If `Object.is(x, y)` and `Object.is(y, z)` are **true**, then `Object.is(x, z)` is also **true**, for any values `x`, `y` and `z`.

SameValueZero

It behaves like SameValue, but considers `+0` and `-0` to be equal.

You can use this comparison algorithm via `Array.prototype.includes` (ECMAScript 7).

Examples:

```
[1].includes(1);            // true
[+0].includes(-0);          // true
[NaN].includes(NaN);        // true
[true].includes("true");    // false
[false].includes(0);        // false
[1].includes("1");          // false
[null].includes(undefined); // false
[[]].includes([]);          // false
```

This algorithm still has the properties of an equivalence relation:

- Reflexivity: `[x].includes(x)` is **true**, for any value `x`
- Symmetry: `[x].includes(y)` is **true** if, and only if, `[y].includes(x)` is **true**, for any values `x` and `y`.
- Transitivity: If `[x].includes(y)` and `[y].includes(z)` are **true**, then `[x].includes(z)` is also **true**, for any values `x`, `y` and `z`.

Strict Equality Comparison

It behaves like SameValue, but

- Considers `+0` and `-0` to be equal.
- Considers **NaN** different than any value, including itself

You can use this comparison algorithm via the `===` operator (ECMAScript 3).

There is also the `!==` operator (ECMAScript 3), which negates the result of `===`.

Examples:

```
1 === 1;            // true
+0 === -0;          // true
NaN === NaN;        // false
true === "true";    // false
false === 0;        // false
1 === "1";          // false
null === undefined; // false
[] === [];          // false
```

This algorithm has the following properties:

- Symmetry: `x === y` is **true** if, and only if, y === `xistrue, for any valuesxandy`.
- Transitivity: If `x === y` and `y === z` are **true**, then `x === z` is also **true**, for any values x, y and z.

But is not an equivalence relation because

- **NaN** is not reflexive: **NaN** `!==` **NaN**

Abstract Equality Comparison

If both operands belong to the same Type, it behaves like the Strict Equality Comparison.

Otherwise, it coerces them as follows:

- **undefined** and **null** are considered to be equal
- When comparing a number with a string, the string is coerced to a number
- When comparing a boolean with something else, the boolean is coerced to a number
- When comparing an object with a number, string or symbol, the object is coerced to a primitive

If there was a coercion, the coerced values are compared recursively. Otherwise the algorithm returns **false**.

You can use this comparison algorithm via the `==` operator (ECMAScript 1).

There is also the `!=` operator (ECMAScript 1), which negates the result of `==`.

Examples:

```
1 == 1;              // true
+0 == -0;            // true
NaN == NaN;          // false
true == "true";      // false
false == 0;          // true
1 == "1";            // true
null == undefined;   // true
[] == [];            // false
```

This algorithm has the following property:

- Symmetry: `x == y` is **true** if, and only if, `y == x` is **true**, for any values x and y.

But is not an equivalence relation because

- **NaN** is not reflexive: **NaN** `!=` **NaN**
- Transitivity does not hold, e.g. `0 == ''` and `0 == '0'`, but `'' != '0'`

Section 10.11: Relational operators (<, <=, >, >=)

When both operands are numeric, they are compared normally:

```
1 < 2         // true
2 <= 2        // true
3 >= 5        // false
true < false // false (implicitly converted to numbers, 1 > 0)
```

When both operands are strings, they are compared lexicographically (according to alphabetical order):

```
'a' < 'b'     // true
'1' < '2'     // true
```

```
'100' > '12' // false ('100' is less than '12' lexicographically!)
```

When one operand is a string and the other is a number, the string is converted to a number before comparison:

```
'1' < 2      // true
'3' > 2      // true
true > '2'   // false (true implicitly converted to number, 1 < 2)
```

When the string is non-numeric, numeric conversion returns `NaN` (not-a-number). Comparing with `NaN` always returns `false`:

```
1 < 'abc'    // false
1 > 'abc'    // false
```

But be careful when comparing a numeric value with `null`, `undefined` or empty strings:

```
1 > ''         // true
1 < ''         // false
1 > null       // true
1 < null       // false
1 > undefined // false
1 < undefined // false
```

When one operand is a object and the other is a number, the object is converted to a number before comparison.So `null` is particular case because `Number(null);//0`

```
new Date(2015)< 1479480185280            // true
null > -1                                //true
({toString:function(){return 123}}) > 122  //true
```

Section 10.12: Inequality

Operator `!=` is the inverse of the `==` operator.
Will return `true` if the operands aren't equal.
The JavaScript engine will try and convert both operands to matching types if they aren't of the same type. **Note:** if the two operands have different internal references in memory, then `false` will be returned.

Sample:

```
1 != '1'     // false
1 != 2       // true
```

In the sample above, `1 != '1'` is `false` because, a primitive number type is being compared to a `char` value. Therefore, the JavaScript engine doesn't care about the datatype of the R.H.S value.

Operator: `!==` is the inverse of the `===` operator. Will return true if the operands are not equal or if their types do not match.

Example:

```
1 !== '1'    // true
1 !== 2      // true
1 !== 1      // false
```

Section 10.13: List of Comparison Operators

Operator	Comparison	Example
`==`	Equal	`i == 0`
`===`	Equal Value and Type	`i === "5"`
`!=`	Not Equal	`i != 5`
`!==`	Not Equal Value or Type	`i !== 5`
`>`	Greater than	`i > 5`
`<`	Less than	`i < 5`
`>=`	Greater than or equal	`i >= 5`
`<=`	Less than or equal	`i <= 5`

Section 10.14: Grouping multiple logic statements

You can group multiple boolean logic statements within parenthesis in order to create a more complex logic evaluation, especially useful in if statements.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

We could also move the grouped logic to variables to make the statement a bit shorter and descriptive:

```
var isLegal = age >= 18;
var tall = height >= 5.11;
var suitable = isLegal && tall;
var isRoyalty = status === 'royalty';
var specialCase = isRoyalty && hasInvitation;
var canEnterOurBar = suitable || specialCase;

if (canEnterOurBar) console.log('You can enter our club');
```

Notice that in this particular example (and many others), grouping the statements with parenthesis works the same as if we removed them, just follow a linear logic evaluation and you'll find yourself with the same result. I do prefer using parenthesis as it allows me to understand clearer what I intended and might prevent for logic mistakes.

Section 10.15: Bit fields to optimise comparison of multi state data

A bit field is a variable that holds various boolean states as individual bits. A bit on would represent true, and off would be false. In the past bit fields were routinely used as they saved memory and reduced processing load. Though the need to use bit field is no longer so important they do offer some benefits that can simplify many processing tasks.

For example user input. When getting input from a keyboard's direction keys up, down, left, right you can encode the various keys into a single variable with each direction assigned a bit.

Example reading keyboard via bitfield

```
var bitField = 0;  // the value to hold the bits
const KEY_BITS = [4,1,8,2]; // left up right down
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // left up right down
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
```

```
        if(e.type === "keydown"){
            bitField |= KEY_BITS[e.keyCode - 37];
        }else{
            bitField &= KEY_MASKS[e.keyCode - 37];
        }
    }
}
```

Example reading as an array

```
var directionState = [false,false,false,false];
window.onkeydown = window.onkeyup = function (e) {
    if(e.keyCode >= 37 && e.keyCode <41){
        directionState[e.keyCode - 37] = e.type === "keydown";
    }
}
```

To turn on a bit use bitwise *or* `|` and the value corresponding to the bit. So if you wish to set the 2nd bit `bitField |= 0b10` will turn it on. If you wish to turn a bit off use bitwise *and* `&` with a value that has all by the required bit on. Using 4 bits and turning the 2nd bit off `bitfield &= 0b1101;`

You may say the above example seems a lot more complex than assigning the various key states to an array. Yes, it is a little more complex to set but the advantage comes when interrogating the state.

If you want to test if all keys are up.

```
// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

You can set some constants to make things easier

```
// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right
```

You can then quickly test for many various keyboard states

```
if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield  & KEY_UL) {             // is Up left down
if ((bitfield & KEY_U) === KEY_U) {   // is Up only down
if (bitfield & KEY_U) {               // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) {            // is Up up (any other key may be down)
if (!bitfield ) {                     // no keys are down
if (bitfield ) {                      // any one or more keys are down
```

The keyboard input is just one example. Bitfields are useful when you have various states that must in combination be acted on. JavaScript can use up to 32 bits for a bit field. Using them can offer significant performance increases. They are worth being familiar with.

Chapter 11: Conditions

Conditional expressions, involving keywords such as if and else, provide JavaScript programs with the ability to perform different actions depending on a Boolean condition: true or false. This section covers the use of JavaScript conditionals, Boolean logic, and ternary statements.

Section 11.1: Ternary operators

Can be used to shorten if/else operations. This comes in handy for returning a value quickly (i.e. in order to assign it to another variable).

For example:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

In this case, `result` gets the 'cute' value, because the value of animal is 'kitty'. If animal had another value, result would get the 'still nice' value.

Compare this to what the code would like with `if/else` conditions.

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

The `if` or `else` conditions may have several operations. In this case the operator returns the result of the last expression.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Because `a` was equal to 0, it becomes `1`, and `str` becomes 'not a test'. The operation which involved `str` was the last, so `b` receives the result of the operation, which is the value contained in `str`, i.e. 'not a test'.

Ternary operators *always* expect else conditions, otherwise you'll get a syntax error. As a workaround you could return a zero something similar in the else branch - this doesn't matter if you aren't using the return value but just shortening (or attempting to shorten) the operation.

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

As you see, `if (a === 1) alert('Hey, it is 1!');` would do the same thing. It would be just a char longer, since it doesn't need an obligatory `else` condition. If an `else` condition was involved, the ternary method would be much cleaner.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

Ternaries can be nested to encapsulate additional logic. For example

```
foo ? bar ? 1 : 2 : 3

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

foo ? (bar ? 1 : 2) : 3
```

This is the same as the following `if`/`else`

```
if (foo) {
  if (bar) {
    1
  } else {
    2
  }
} else {
  3
}
```

Stylistically this should only be used with short variable names, as multi-line ternaries can drastically decrease readability.

The only statements which cannot be used in ternaries are control statements. For example, you cannot use return or break with ternaries. The following expression will be invalid.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
    (animal === 'kitty') ? break:console.log(i);
}
```

For return statements, the following would also be invalid:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

To do the above properly, you would return the ternary as follows:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

Section 11.2: Switch statement

Switch statements compare the value of an expression against 1 or more values and executes different sections of code based on that comparison.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

The **break** statement "breaks" out of the switch statement and ensures no more code within the switch statement is executed. This is how sections are defined and allows the user to make "fall through" cases.

> **Warning**: lack of a **break** or **return** statement for each case means the program will continue to evaluate the next case, even if the case criteria is unmet!

```
switch (value) {
  case 1:
    console.log('I will only run if value === 1');
    // Here, the code "falls through" and will run the code under case 2
  case 2:
    console.log('I will run if value === 1 or value === 2');
    break;
  case 3:
    console.log('I will only run if value === 3');
    break;
}
```

The last case is the **default** case. This one will run if no other matches were made.

```
var animal = 'Lion';
switch (animal) {
  case 'Dog':
    console.log('I will not run since animal !== "Dog"');
    break;
  case 'Cat':
    console.log('I will not run since animal !== "Cat"');
    break;
  default:
    console.log('I will run since animal does not match any other case');
}
```

It should be noted that a case expression can be any kind of expression. This means you can use comparisons, function calls, etc. as case values.

```
function john() {
  return 'John';
}

function jacob() {
  return 'Jacob';
}

switch (name) {
  case john(): // Compare name with the return value of john() (name == "John")
    console.log('I will run if name === "John"');
    break;
  case 'Ja' + 'ne': // Concatenate the strings together then compare (name == "Jane")
    console.log('I will run if name === "Jane"');
    break;
  case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
    console.log('His name is equal to name too!');
    break;
}
```

Multiple Inclusive Criteria for Cases

Since cases "fall through" without a **break** or **return** statement, you can use this to create multiple inclusive criteria:

```
var x = "c"
switch (x) {
   case "a":
   case "b":
   case "c":
      console.log("Either a, b, or c was selected.");
      break;
   case "d":
      console.log("Only d was selected.");
      break;
   default:
      console.log("No case was matched.");
      break;  // precautionary break if case order changes
}
```

Section 11.3: If / Else If / Else Control

In its most simple form, an `if` condition can be used like this:

```
var i = 0;

if (i < 1) {
    console.log("i is smaller than 1");
}
```

The condition `i < 1` is evaluated, and if it evaluates to **`true`** the block that follows is executed. If it evaluates to **`false`**, the block is skipped.

An `if` condition can be expanded with an **`else`** block. The condition is checked *once* as above, and if it evaluates to **`false`** a secondary block will be executed (which would be skipped if the condition were **`true`**). An example:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    console.log("i was not smaller than 1");
}
```

Supposing the **`else`** block contains nothing but another `if` block (with optionally an **`else`** block) like this:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else {
    if (i < 2) {
        console.log("i is smaller than 2");
    } else {
        console.log("none of the previous conditions was true");
    }
}
```

Then there is also a different way to write this which reduces nesting:

```
if (i < 1) {
    console.log("i is smaller than 1");
} else if (i < 2) {
    console.log("i is smaller than 2");
} else {
    console.log("none of the previous conditions was true");
```

```
}
```

Some important footnotes about the above examples:

- If any one condition evaluated to `true`, no other condition in that chain of blocks will be evaluated, and all corresponding blocks (including the `else` block) will not be executed.
- The number of `else if` parts is practically unlimited. The last example above only contains one, but you can have as many as you like.
- The *condition* inside an `if` statement can be anything that can be coerced to a boolean value, see the topic on boolean logic for more details;
- The `if-else-if` ladder exits at the first success. That is, in the example above, if the value of `i` is 0.5 then the first branch is executed. If the conditions overlap, the first criteria occurring in the flow of execution is executed. The other condition, which could also be true is ignored.
- If you have only one statement, the braces around that statement are technically optional, e.g this is fine:

  ```
  if (i < 1) console.log("i is smaller than 1");
  ```

  And this will work as well:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
  ```

  If you want to execute multiple statements inside an `if` block, then the curly braces around them are mandatory. Only using indentation isn't enough. For example, the following code:

  ```
  if (i < 1)
      console.log("i is smaller than 1");
      console.log("this will run REGARDLESS of the condition"); // Warning, see text!
  ```

  is equivalent to:

  ```
  if (i < 1) {
      console.log("i is smaller than 1");
  }
  console.log("this will run REGARDLESS of the condition");
  ```

Section 11.4: Strategy

A strategy pattern can be used in JavaScript in many cases to replace a switch statement. It is especially helpful when the number of conditions is dynamic or very large. It allows the code for each condition to be independent and separately testable.

Strategy object is simple an object with multiple functions, representing each separate condition. Example:

```
const AnimalSays = {
    dog () {
        return 'woof';
    },

    cat () {
        return 'meow';
    },
```

```
    lion () {
        return 'roar';
    },

    // ... other animals

    default () {
        return 'moo';
    }
};
```

The above object can be used as follows:

```
function makeAnimalSpeak (animal) {
    // Match the animal by type
    const speak = AnimalSays[animal] || AnimalSays.default;
    console.log(animal + ' says ' + speak());
}
```

Results:

```
makeAnimalSpeak('dog') // => 'dog says woof'
makeAnimalSpeak('cat') // => 'cat says meow'
makeAnimalSpeak('lion') // => 'lion says roar'
makeAnimalSpeak('snake') // => 'snake says moo'
```

In the last case, our default function handles any missing animals.

Section 11.5: Using || and && short circuiting

The Boolean operators `||` and `&&` will "short circuit" and not evaluate the second parameter if the first is true or false respectively. This can be used to write short conditionals like:

```
var x = 10

x == 10 && alert("x is 10")
x == 10 || alert("x is not 10")
```

Chapter 12: Arrays

Section 12.1: Converting Array-like Objects to Arrays

What are Array-like Objects?

JavaScript has "Array-like Objects", which are Object representations of Arrays with a length property. For example:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Common examples of Array-like Objects are the `arguments` object in functions and `HTMLCollection` or `NodeList` objects returned from methods like `document.getElementsByTagName` or `document.querySelectorAll`.

However, one key difference between Arrays and Array-like Objects is that Array-like objects inherit from `Object.prototype` instead of `Array.prototype`. This means that Array-like Objects can't access common Array prototype methods like `forEach()`, `push()`, `map()`, `filter()`, and `slice()`:

```
var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3);   // → [1, 2, 3]
```

Convert Array-like Objects to Arrays in ES6

1. `Array.from`:

Version ≥ 6

```
const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => {/* Do something */}); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => {/* Do something */}); // Works
```

2. `for...of`:

Version ≥ 6

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. Spread operator:

Version ≥ 6

```
[...arrayLike]
```

4. `Object.values`:

Version ≥ 7

```
var realArray = Object.values(arrayLike);
```

5. `Object.keys`:

Version ≥ 6

```
var realArray = Object
   .keys(arrayLike)
   .map((key) => arrayLike[key]);
```

Convert Array-like Objects to Arrays in ≤ ES5

Use `Array.prototype.slice` like so:

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

You can also use `Function.prototype.call` to call `Array.prototype` methods on Array-like objects directly, without converting them:

Version ≥ 5.1

```
var domList = document.querySelectorAll('#myDropdown option');

domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

You can also use `[].method.bind( arrayLikeObject )` to borrow array methods and glom them on to your object:

Version ≥ 5.1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};

arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

[].forEach.bind(arrayLike)(function(val){
  // Do stuff with val
```

```
}); // Wow! this works
```
Modifying Items During Conversion

In ES6, while using `Array.from`, we can specify a map function that returns a mapped value for the new array being created.

Version ≥ 6
```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

See Arrays are Objects for a detailed analysis.

Section 12.2: Reducing values

Version ≥ 5.1

The `reduce()` method applies a function against an accumulator and each value of the array (from left-to-right) to reduce it to a single value.

Array Sum

This method can be used to condense all values of an array into a single value:

```
[1, 2, 3, 4].reduce(function(a, b) {
  return a + b;
});
// → 10
```

Optional second parameter can be passed to `reduce()`. Its value will be used as the first argument (specified as `a`) for the first call to the callback (specified as `function(a, b)`).

```
[2].reduce(function(a, b) {
  console.log(a, b); // prints: 1 2
  return a + b;
}, 1);
// → 3
```

Version ≥ 5.1
Flatten Array of Objects

The example below shows how to flatten an array of objects into a single object.

```
var array = [{
    key: 'one',
    value: 1
}, {
    key: 'two',
    value: 2
}, {
    key: 'three',
    value: 3
}];
```
Version ≤ 5.1
```
array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

Version ≥ 6

```
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
}), {});
```

Version ≥ 7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Note that the Rest/Spread Properties is not in the list of finished proposals of ES2016. It isn't supported by ES2016. But we can use babel plugin babel-plugin-transform-object-rest-spread to support it.

All of the above examples for Flatten Array result in:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

Version ≥ 5.1

Map Using Reduce

As another example of using the *initial value* parameter, consider the task of calling a function on an array of items, returning the results in a new array. Since arrays are ordinary values and list concatenation is an ordinary function, we can use `reduce` to accumulate a list, as the following example demonstrates:

```
function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// Usage:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]
```

Note that this is for illustration (of the initial value parameter) only, use the native `map` for working with list transformations (see Mapping values for the details).

Version ≥ 5.1

Find Min or Max Value

We can use the accumulator to keep track of an array element as well. Here is an example leveraging this to find the min value:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

Version ≥ 6

Find Unique Values

Here is an example that uses reduce to return the unique numbers to an array. An empty array is passed as the second argument and is referenced by `prev`.

```
var arr = [1, 2, 1, 5, 9, 5];

arr.reduce((prev, number) => {
  if(prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

Section 12.3: Mapping values

It is often necessary to generate a new array based on the values of an existing array.

For example, to generate an array of string lengths from an array of strings:

Version ≥ 5.1

```
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]
```

Version ≥ 6

```
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

In this example, an anonymous function is provided to the `map()` function, and the map function will call it for every element in the array, providing the following parameters, in this order:

- The element itself
- The index of the element (0, 1...)
- The entire array

Additionally, `map()` provides an *optional* second parameter in order to set the value of **`this`** in the mapping function. Depending on the execution environment, the default value of **`this`** might vary:

In a browser, the default value of **`this`** is always `window`:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

You can change it to any custom object like this:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Section 12.4: Filtering Object Arrays

The `filter()` method accepts a test function, and returns a new array containing only the elements of the original array that pass the test provided.

```
// Suppose we want to get all odd number in an array:
var numbers = [5, 32, 43, 4];
```

Version ≥ 5.1

```
var odd = numbers.filter(function(n) {
  return n % 2 !== 0;
});
```

Version ≥ 6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

`odd` would contain the following array: `[5, 43]`.

It also works on an array of objects:

```
var people = [{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}, {
  id: 3,
  name: "Peter",
  age: 55
}];
```

Version ≥ 5.1

```
var young = people.filter(function(person) {
  return person.age < 35;
});
```

Version ≥ 6

```
let young = people.filter(person => person.age < 35);
```

`young` would contain the following array:

```
[{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}]
```

You can search in the whole array for a value like this:

```
var young = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
    if(String(val).indexOf("J") > -1) {
      flag = true;
      return;
    }
  });
  if(flag) return obj;
});
```

This returns:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Section 12.5: Sorting Arrays

The `.sort()` method sorts the elements of an array. The default method will sort the array according to string Unicode code points. To sort an array numerically the `.sort()` method needs to have a `compareFunction` passed to it.

> **Note:** The `.sort()` method is impure. `.sort()` will sort the array **in-place**, i.e., instead of creating a sorted copy of the original array, it will re-order the original array and return it.

Default Sort

Sorts the array in UNICODE order.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Results in:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

> **Note:** The uppercase characters have moved above lowercase. The array is not in alphabetical order, and numbers are not in numerical order.

Alphabetical Sort

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
  return a.localeCompare(b);
});
```

Results in:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

> **Note:** The above sort will throw an error if any array items are not a string. If you know that the array may contain items that are not strings use the safe version below.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {
  return a.toString().localeCompare(b);
});
```

String sorting by length (longest first)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return b.length - a.length;
});
```

Results in

```
["elephants", "penguins", "zebras", "dogs"];
```

String sorting by length (shortest first)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {
  return a.length - b.length;
});
```

Results in

```
["dogs", "zebras", "penguins", "elephants"];
```

Numerical Sort (ascending)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return a - b;
});
```

Results in:

```
[1, 10, 100, 1000, 10000]
```

Numerical Sort (descending)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {
  return b - a;
});
```

Results in:

```
[10000, 1000, 100, 10, 1]
```

Sorting array by even and odd numbers

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {
    return (a & 1) - (b & 1) || a - b;
});
```

Results in:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Date Sort (descending)

```
var dates = [
  new Date(2007, 11, 10),
  new Date(2014, 2, 21),
```

```
  new Date(2009, 6, 11),
  new Date(2016, 7, 23)
];

dates.sort(function(a, b) {
  if (a > b) return -1;
  if (a < b) return 1;
  return 0;
});

// the date objects can also sort by its difference
// the same way that numbers array is sorting
dates.sort(function(a, b) {
  return b-a;
});
```

Results in:

```
[
  "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",
  "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",
  "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",
  "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"
]
```

Section 12.6: Iteration

A traditional `for`-loop

A traditional `for` loop has three components:

1. **The initialization:** executed before the look block is executed the first time
2. **The condition:** checks a condition every time before the loop block is executed, and quits the loop if false
3. **The afterthought:** performed every time after the loop block is executed

These three components are separated from each other by a `;` symbol. Content for each of these three components is optional, which means that the following is the most minimal `for` loop possible:

```
for (;;) {
    // Do stuff
}
```

Of course, you will need to include an `if(condition === true) { break; }` or an `if(condition === true) { return; }` somewhere inside that `for`-loop to get it to stop running.

Usually, though, the initialization is used to declare an index, the condition is used to compare that index with a minimum or maximum value, and the afterthought is used to increment the index:

```
for (var i = 0, length = 10; i < length; i++) {
    console.log(i);
}
```

Using a traditional `for` loop to loop through an array

The traditional way to loop through an array, is this:

```
for (var i = 0, length = myArray.length; i < length; i++) {
```

```
    console.log(myArray[i]);
}
```

Or, if you prefer to loop backwards, you do this:

```
for (var i = myArray.length - 1; i > -1; i--) {
    console.log(myArray[i]);
}
```

There are, however, many variations possible, like for example this one:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value =
myArray[++key]) {
    console.log(value);
}
```

... or this one ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
    console.log(myArray[i]);
    i++;
}
```

... or this one:

```
var key = 0, value;
for (; value = myArray[key++];){
    console.log(value);
}
```

Whichever works best is largely a matter of both personal taste and the specific use case you're implementing.

Note that each of these variations is supported by all browsers, including very very old ones!

A `while` loop

One alternative to a `for` loop is a `while` loop. To loop through an array, you could do this:

```
var key = 0;
while(value = myArray[key++]){
    console.log(value);
}
```

Like traditional `for` loops, `while` loops are supported by even the oldest of browsers.

Also, note that every while loop can be rewritten as a `for` loop. For example, the `while` loop hereabove behaves the exact same way as this `for`-loop:

```
for(var key = 0; value = myArray[key++];){
    console.log(value);
}
```

`for...in`

In JavaScript, you can also do this:

```
for (i in myArray) {
    console.log(myArray[i]);
}
```

This should be used with care, however, as it doesn't behave the same as a traditional **for** loop in all cases, and there are potential side-effects that need to be considered. See **Why is using "for...in" with array iteration a bad idea?** for more details.

for...of

In ES 6, the `for-of` loop is the recommended way of iterating over a the values of an array:

Version ≥ 6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

The following example shows the difference between a `for...of` loop and a `for...in` loop:

Version ≥ 6

```
let myArray = [3, 5, 7];
myArray.foo = "hello";

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

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

Array.prototype.keys()

The `Array.prototype.keys()` method can be used to iterate over indices like this:

Version ≥ 6

```
let myArray = [1, 2, 3, 4];
for (let i of myArray.keys()) {
  let twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

Array.prototype.forEach()

The `.forEach(...)` method is an option in ES 5 and above. It is supported by all modern browsers, as well as Internet Explorer 9 and later.

Version ≥ 5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {
  var twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
});
```

Comparing with the traditional **for** loop, we can't jump out of the loop in `.forEach()`. In this case, use the **for** loop, or use partial iteration presented below.

In all versions of JavaScript, it is possible to iterate through the indices of an array using a traditional C-style `for` loop.

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
  var twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

It's also possible to use `while` loop:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
  sum += i;
}
console.log(sum);
```

`Array.prototype.every`

Since ES5, if you want to iterate over a portion of an array, you can use [`Array.prototype.every`](), which iterates until we return `false`:

Version ≥ 5

```
// [].every() stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].every(function(value, index, arr) {
  console.log(value);
  return value % 2 === 0; // iterate until an odd number is found
});
```

Equivalent in any JavaScript version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is
found
  console.log(arr[i]);
}
```

`Array.prototype.some`

[`Array.prototype.some`]() iterates until we return `true`:

Version ≥ 5

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
  console.log(value);
  return value === 7; // iterate until we find value 7
});
```

Equivalent in any JavaScript version:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
  console.log(arr[i]);
}
```

Libraries

Finally, many utility libraries also have their own `foreach` variation. Three of the most popular ones are these:

[`jQuery.each()`](), in [jQuery]():

```
$.each(myArray, function(key, value) {
    console.log(value);
});
```

[`_.each()`](), in [Underscore.js]():

```
_.each(myArray, function(value, key, myArray) {
    console.log(value);
});
```

[`_.forEach()`](), in [Lodash.js]():

```
_.forEach(myArray, function(value, key) {
    console.log(value);
});
```

See also the following question on SO, where much of this information was originally posted:

- [Loop through an array in JavaScript]()

Section 12.7: Destructuring an array

`Version ≥ 6`

An array can be destructured when being assigned to a new variable.

```
const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3;     // → true
height === 4;     // → true
hypotneuse === 5; // → true
```

Elements can be skipped

```
const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4
```

Rest operator can be used too

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

An array can also be destructured if it's an argument to a function.

```
function area([length, height]) {
  return (length * height) / 2;
}
```

```
const triangle = [3, 4, 5];

area(triangle); // → 6
```

Notice the third argument is not named in the function because it's not needed.

Learn more about destructuring syntax.

Section 12.8: Removing duplicate elements

From ES5.1 onwards, you can use the native method `Array.prototype.filter` to loop through an array and leave only entries that pass a given callback function.

In the following example, our callback checks if the given value occurs in the array. If it does, it is a duplicate and will not be copied to the resulting array.

Version ≥ 5.1

```
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

If your environment supports ES6, you can also use the Set object. This object lets you store unique values of any type, whether primitive values or object references:

Version ≥ 6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Section 12.9: Array comparison

For simple array comparison you can use JSON stringify and compare the output strings:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

> **Note:** that this will only work if both objects are JSON serializable and do not contain cyclic references. It may throw `TypeError: Converting circular structure to JSON`

You can use a recursive function to compare arrays.

```
function compareArrays(array1, array2) {
  var i, isA1, isA2;
  isA1 = Array.isArray(array1);
  isA2 = Array.isArray(array2);

  if (isA1 !== isA2) { // is one an array and the other not?
    return false;      // yes then can not be the same
  }
  if (! (isA1 && isA2)) {      // Are both not arrays
    return array1 === array2;  // return strict equality
  }
  if (array1.length !== array2.length) { // if lengths differ then can not be the same
    return false;
  }
  // iterate arrays and compare them
  for (i = 0; i < array1.length; i += 1) {
```

```
    if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
      return false;
    }
  }
  return true; // must be equal
}
```

WARNING: Using the above function is dangerous and should be wrapped in a `try` `catch` if you suspect there is a chance the array has cyclic references (a reference to an array that contains a reference to itself)

```
a = [0] ;
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded
```

> **Note:** The function uses the strict equality operator === to compare non array items `{a: 0} === {a: 0}` is `false`

Section 12.10: Reversing arrays

`.reverse` is used to reverse the order of items inside an array.

Example for `.reverse`:

```
[1, 2, 3, 4].reverse();
```

Results in:

```
[4, 3, 2, 1]
```

> **Note**: Please note that `.reverse`(`Array.prototype.reverse`) will reverse the array *in place*. Instead of returning a reversed copy, it will return the same array, reversed.
>
> ```
> var arr1 = [11, 22, 33];
> var arr2 = arr1.reverse();
> console.log(arr2); // [33, 22, 11]
> console.log(arr1); // [33, 22, 11]
> ```

You can also reverse an array 'deeply' by:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Example for deepReverse:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
```

```
deepReverse(arr);
```

Results in:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Section 12.11: Shallow cloning an array

Sometimes, you need to work with an array while ensuring you don't modify the original. Instead of a `clone` method, arrays have a `slice` method that lets you perform a shallow copy of any part of an array. Keep in mind that this only clones the first level. This works well with primitive types, like numbers and strings, but not objects.

To shallow-clone an array (i.e. have a new array instance but with the same elements), you can use the following one-liner:

```
var clone = arrayToClone.slice();
```

This calls the built-in JavaScript `Array.prototype.slice` method. If you pass arguments to `slice`, you can get more complicated behaviors that create shallow clones of only part of an array, but for our purposes just calling `slice()` will create a shallow copy of the entire array.

All method used to convert array like objects to array are applicable to clone an array:

Version ≥ 6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

Version ≤ 5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Section 12.12: Concatenating Arrays

Two Arrays

```
var array1 = [1, 2];
var array2 = [3, 4, 5];
```

Version ≥ 3

```
var array3 = array1.concat(array2);  // returns a new array
```

Version ≥ 6

```
var array3 = [...array1, ...array2]
```

Results in a new `Array`:

```
[1, 2, 3, 4, 5]
```

Multiple Arrays

```
var array1 = ["a", "b"],
    array2 = ["c", "d"],
    array3 = ["e", "f"],
    array4 = ["g", "h"];
```

Version ≥ 3

Provide more Array arguments to `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

Version ≥ 6

Provide more arguments to `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Results in a new `Array`:

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Without Copying the First Array

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],
    shortArray = [9, 10];
```

Version ≥ 3

Provide the elements of `shortArray` as parameters to push using `Function.prototype.apply`

```
longArray.push.apply(longArray, shortArray);
```

Version ≥ 6

Use the spread operator to pass the elements of `shortArray` as separate arguments to `push`

```
longArray.push(...shortArray)
```

The value of `longArray` is now:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note that if the second array is too long (>100,000 entries), you may get a stack overflow error (because of how `apply` works). To be safe, you can iterate instead:

```
shortArray.forEach(function (elem) {
    longArray.push(elem);
});
```

Array and non-array values

```
var array = ["a", "b"];
```

Version ≥ 3

```
var arrConc = array.concat("c", "d");
```

Version ≥ 6

```
var arrConc = [...array, "c", "d"]
```

Results in a new `Array`:

```
["a", "b", "c", "d"]
```

You can also mix arrays with non-arrays

```
var arr1 = ["a","b"];
var arr2 = ["e", "f"];

var arrConc = arr1.concat("c", "d", arr2);
```

Results in a new `Array`:

```
["a", "b", "c", "d", "e", "f"]
```

Section 12.13: Merge two array as key value pair

When we have two separate array and we want to make key value pair from that two array, we can use array's reduce function like below:

```
var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result =  rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {})

console.log(result);
```

Output:

```
{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}
```

Section 12.14: Array spread / rest

Spread operator
`Version ≥ 6`

With ES6, you can use spreads to separate individual elements into a comma-separated syntax:

```
let arr = [1, 2, 3, ...[4, 5, 6]];  // [1, 2, 3, 4, 5, 6]

// in ES < 6, the operations above are equivalent to
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

The spread operator also acts upon strings, separating each individual character into a new string element. Therefore, using an array function for converting these into integers, the array created above is equivalent to the one below:

```
let arr = [1, 2, 3, ...[..."456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

Or, using a single string, this could be simplified to:

```
let arr = [..."123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

If the mapping is not performed then:

```
let arr = [..."123456"]; // ["1", "2", "3", "4", "5", "6"]
```

The spread operator can also be used to spread arguments into a function:

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];

myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Rest operator

The rest operator does the opposite of the spread operator by coalescing several elements into a single one

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Collect arguments of a function:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Section 12.15: Filtering values

The `filter()` method creates an array filled with all array elements that pass a test provided as a function.

Version ≥ 5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

Version ≥ 6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Results in a new array:

```
[3, 4, 5]
```

Filter falsy values

Version ≥ 5.1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Since Boolean is a native JavaScript function/constructor that takes [one optional parameter] and the filter method also takes a function and passes it the current array item as parameter, you could read it like the following:

1. `Boolean(0)` returns false
2. `Boolean(undefined)` returns false
3. `Boolean({})` returns **true** which means push it to the returned array
4. `Boolean(null)` returns false
5. `Boolean('')` returns false
6. `Boolean(true)` returns **true** which means push it to the returned array
7. `Boolean(5)` returns **true** which means push it to the returned array

so the overall process will result

```
[ {}, true, 5 ]
```

Another simple example

This example utilises the same concept of passing a function that takes one argument

Version ≥ 5.1

```
function startsWithLetterA(str) {
    if(str && str[0].toLowerCase() == 'a') {
        return true
    }
    return false;
}

var str              = 'Since Boolean is a native javascript function/constructor that takes [one optional parameter] and the filter method also takes a function and passes it the current array item as a parameter, you could read it like the following';
var strArray         = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//["a", "and", "also", "a", "and", "array", "as"]
```

Section 12.16: Searching an Array

The recommended way (Since ES5) is to use Array.prototype.find:

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

In any version of JavaScript, a standard **for** loop can be used as well:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

The findIndex() method returns an index in the array, if an element in the array satisfies the provided testing function. Otherwise -1 is returned.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
```

```
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Section 12.17: Convert a String to an Array

The `.split()` method splits a string into an array of substrings. By default `.split()` will break the string into substrings on spaces (`" "`), which is equivalent to calling `.split(" ")`.

The parameter passed to `.split()` specifies the character, or the regular expression, to use for splitting the string.

To split a string into an array call `.split` with an empty string (`""`). **Important Note:** This only works if all of your characters fit in the Unicode lower range characters, which covers most English and most European languages. For languages that require 3 and 4 byte Unicode characters, `slice("")` will separate them.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

Version ≥ 6

Using the spread operator (`...`), to convert a `string` into an `array`.

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Section 12.18: Removing items from an array

Shift

Use `.shift` to remove the first item of an array.

For example:

```
var array = [1, 2, 3, 4];
array.shift();
```

array results in:

```
[2, 3, 4]
```

Pop

Further `.pop` is used to remove the last item from an array.

For example:

```
var array = [1, 2, 3];
array.pop();
```

array results in:

```
[1, 2]
```

Both methods return the removed item;

Splice

Use `.splice()` to remove a series of elements from an array. `.splice()` accepts two parameters, the starting index, and an optional number of elements to delete. If the second parameter is left out `.splice()` will remove all elements from the starting index through the end of the array.

For example:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

leaves `array` containing:

```
[1, 4]
```

The return of `array.splice()` is a new array containing the removed elements. For the example above, the return would be:

```
[2, 3]
```

Thus, omitting the second parameter effectively splits the array into two arrays, with the original ending before the index specified:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

...leaves `array` containing `[1, 2]` and returns `[3, 4]`.

Delete

Use **`delete`** to remove item from array without changing the length of array:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

Assigning value to `length` of array changes the length to given value. If new value is less than array length items will be removed from the end of value.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Section 12.19: Removing all elements

```
var arr = [1, 2, 3, 4];
```

Method 1

Creates a new array and overwrites the existing array reference with a new one.

```
arr = [];
```

Care must be taken as this does not remove any items from the original array. The array may have been closed over when passed to a function. The array will remain in memory for the life of the function though you may not be aware of this. This is a common source of memory leaks.

Example of a memory leak resulting from bad array clearing:

```
var count = 0;

function addListener(arr) { // arr is closed over
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // this functions reference keeps
    // the closure current while the
    // event is active
    // do something but does not need arr
  });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
  addListener(arr); // the array is passed to the function
  arr = []; // only removes the reference, the original array remains
  array.push("some large data"); // more memory allocated
  i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

To prevent the risk of a memory leak use the one of the following 2 methods to empty the array in the above example's while loop.

Method 2

Setting the length property deletes all array element from the new array length to the old array length. It is the most efficient way to remove and dereference all items in the array. Keeps the reference to the original array

```
arr.length = 0;
```

Method 3

Similar to method 2 but returns a new array containing the removed items. If you do not need the items this method is inefficient as the new array is still created only to be immediately dereferenced.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
var keepArr = arr.splice(0); // empties the array and creates a new array containing the
                             // removed items
```

Related question.

Section 12.20: Finding the minimum or maximum element

If your array or array-like object is *numeric*, that is, if all its elements are numbers, then you can use `Math.min.apply` or `Math.max.apply` by passing `null` as the first argument, and your array as the second.

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); // 1
```

```
Math.max.apply(null, myArray); // 4
```

Version ≥ 6

In ES6 you can use the `...` operator to spread an array and take the minimum or maximum element.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

The following example uses a `for` loop:

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
   var currentValue = myArray[i];
   if(currentValue > maxValue) {
      maxValue = currentValue;
   }
}
```

Version ≥ 5.1

The following example uses `Array.prototype.reduce()` to find the minimum or maximum:

```
var myArray = [1, 2, 3, 4];

myArray.reduce(function(a, b) {
  return Math.min(a, b);
}); // 1

myArray.reduce(function(a, b) {
  return Math.max(a, b);
}); // 4
```

Version ≥ 6

or using arrow functions:

```
myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4
```

Version ≥ 5.1

To generalize the `reduce` version we'd have to pass in an *initial value* to cover the empty list case:

```
function myMax(array) {
  return array.reduce(function(maxSoFar, element) {
    return Math.max(maxSoFar, element);
  }, -Infinity);
}

myMax([3, 5]);             // 5
myMax([]);                 // -Infinity
Math.max.apply(null, []);  // -Infinity
```

For the details on how to properly use `reduce` see Reducing values.

Section 12.21: Standard array initialization

There are many ways to create arrays. The most common are to use array literals, or the Array constructor:

```
var arr = [1, 2, 3, 4];
var arr2 = new Array(1, 2, 3, 4);
```

If the Array constructor is used with no arguments, an empty array is created.

```
var arr3 = new Array();
```

results in:

```
[]
```

Note that if it's used with exactly one argument and that argument is a `number`, an array of that length with all **`undefined`** values will be created instead:

```
var arr4 = new Array(4);
```

results in:

```
[undefined, undefined, undefined, undefined]
```

That does not apply if the single argument is non-numeric:

```
var arr5 = new Array("foo");
```

results in:

```
["foo"]
```

Version ≥ 6

Similar to an array literal, `Array.of` can be used to create a new `Array` instance given a number of arguments:

```
Array.of(21, "Hello", "World");
```

results in:

```
[21, "Hello", "World"]
```

In contrast to the Array constructor, creating an array with a single number such as `Array.of(23)` will create a new array `[23]`, rather than an Array with length 23.

The other way to create and initialize an array would be `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

will result:

```
[0, 1, 16, 81, 256]
```

Section 12.22: Joining array elements in a string

To join all of an array's elements into a string, you can use the `join` method:

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
```

```
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

As you can see in the second line, items that are not strings will be converted first.

Section 12.23: Removing/Adding elements using splice()

The `splice()` method can be used to remove elements from an array. In this example, we remove the first `3` from the array.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

The `splice()` method can also be used to add elements to an array. In this example, we will insert the numbers 6, 7, and 8 to the end of the array.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

The first argument of the `splice()` method is the index at which to remove/insert elements. The second argument is the number of elements to remove. The third argument and onwards are the values to insert into the array.

Section 12.24: The entries() method

The `entries()` method returns a new Array Iterator object that contains the key/value pairs for each index in the array.

Version ≥ 6

```
var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
  console.log(index,element);
}
```

result

```
0 "a"
1 "b"
2 "c"
```

Note: This method is not supported in Internet Explorer.

Portions of this content from `Array.prototype.entries` by Mozilla Contributors licensed under CC-by-SA 2.5

Section 12.25: Remove value from array

When you need to remove a specific value from an array, you can use the following one-liner to create a copy array without the given value:

```
array.filter(function(val) { return val !== to_remove; });
```

Or if you want to change the array itself without creating a copy (for example if you write a function that get an array as a function and manipulates it) you can use this snippet:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

And if you need to remove just the first value found, remove the while loop:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index , 1); }
```

Section 12.26: Flattening Arrays

2 Dimensional arrays

Version ≥ 6

In ES6, we can flatten the array by the spread operator `...`:

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

Version ≥ 5

In ES5, we can achieve that by .apply():

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Higher Dimension Arrays

Given a deeply nested array like so

```
var deeplyNested = [4,[5,6,[7,8],9]];
```

It can be flattened with this magic

```
console.log(String(deeplyNested).split(',').map(Number);
#=> [4,5,6,7,8,9]
```

Or

```
const flatten = deeplyNested.toString().split(',').map(Number)
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Both of the above methods only work when the array is made up exclusively of numbers. A multi-dimensional array of objects cannot be flattened by this method.

Section 12.27: Append / Prepend items to Array

Unshift

Use `.unshift` to add one or more items in the beginning of an array.

For example:

```
var array = [3, 4, 5, 6];
array.unshift(1, 2);
```

array results in:

```
[1, 2, 3, 4, 5, 6]
```

Push

Further `.push` is used to add items after the last currently existent item.

For example:

```
var array = [1, 2, 3];
array.push(4, 5, 6);
```

array results in:

```
[1, 2, 3, 4, 5, 6]
```

Both methods return the new array length.

Section 12.28: Object keys and values to array

```
var object = {
    key1: 10,
    key2: 3,
    key3: 40,
    key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Now array is

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Section 12.29: Logical connective of values

Version ≥ 5.1

`.some` and `.every` allow a logical connective of Array values.

While `.some` combines the return values with `OR`, `.every` combines them with `AND`.

Examples for `.some`

```
[false, false].some(function(value) {
  return value;
});
// Result: false

[false, true].some(function(value) {
  return value;
});
// Result: true

[true, true].some(function(value) {
  return value;
});
// Result: true
```

And examples for `.every`

```
[false, false].every(function(value) {
  return value;
});
// Result: false

[false, true].every(function(value) {
  return value;
});
// Result: false

[true, true].every(function(value) {
  return value;
});
// Result: true
```

Section 12.30: Checking if an object is an Array

`Array.isArray(obj)` returns **`true`** if the object is an `Array`, otherwise **`false`**.

```
Array.isArray([])           // true
Array.isArray([1, 2, 3])    // true
Array.isArray({})           // false
Array.isArray(1)            // false
```

In most cases you can **`instanceof`** to check if an object is an `Array`.

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` has the an advantage over using a **`instanceof`** check in that it will still return **`true`** even if the prototype of the array has been changed and will return **`false`** if a non-arrays prototype was changed to the `Array` prototype.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr);   // true
arr instanceof Array; // false
```

Section 12.31: Insert an item into an array at a specific index

Simple item insertion can be done with `Array.prototype.splice` method:

```
arr.splice(index, 0, item);
```

More advanced variant with multiple arguments and chaining support:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
  return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6);  // ["b", "X", "Y", "Z", "c"]
```

And with array-type arguments merging and chaining support:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
  index = Math.min(index, this.length);
  arguments.length > 1
    && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
    && this.insert.apply(this, arguments);
  return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-");  // "a-b-V-W-X-Y-Z-c-d"
```

Section 12.32: Sorting multidimensional array

Given the following array

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

You can sort it sort it by number(second index)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

Version ≥ 6

```
array.sort((a,b) => a[1] - b[1]);
```

This will output

```
[
  ["key2", 3],
  ["key1", 10],
  ["key4", 20],
  ["key3", 40]
]
```

Be aware that the sort method operates on the array *in place*. It changes the array. Most other array methods return a new array, leaving the original one intact. This is especially important to note if you use a functional programming style and expect functions to not have side-effects.

Section 12.33: Test all array items for equality

The `.every` method tests if all array elements pass a provided predicate test.

To test all objects for equality, you can use the following code snippets.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true
```

Version ≥ 6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

The following code snippets test for property equality

```
let data = [
  { name: "alice", id: 111 },
  { name: "alice", id: 222 }
];

data.every(function(item, i, list) { return item === list[0]; }); // false
data.every(function(item, i, list) { return item.name === list[0].name; }); // true
```

Version ≥ 6

```
data.every((item, i, list) => item.name === list[0].name); // true
```

Section 12.34: Copy part of an Array

The slice() method returns a copy of a portion of an array.

It takes two parameters, `arr.slice([begin[, end]])` :

begin

Zero-based index which is the beginning of extraction.

end

Zero-based index which is the end of extraction, slicing up to this index but it's not included.

If the end is a negative number,`end = arr.length + end`.

Example 1

```
// Let's say we have this Array of Alphabets
var arr = ["a", "b", "c", "d"...];

// I want an Array of the first two Alphabets
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Example 2

```
// Let's say we have this Array of Numbers
// and I don't know it's end
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];

// I want to slice this Array starting from
// number 5 to its end
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

Chapter 13: Objects

Property	Description
`value`	The value to assign to the property.
`writable`	Whether the value of the property can be changed or not.
`enumerable`	Whether the property will be enumerated in `for in` loops or not.
`configurable`	Whether it will be possible to redefine the property descriptor or not.
`get`	A function to be called that will return the value of the property.
`set`	A function to be called when the property is assigned a value.

Section 13.1: Shallow cloning

Version ≥ 6

ES6's `Object.assign()` function can be used to copy all of the **enumerable** properties from an existing `Object` instance to a new one.

```
const existing = { a: 1, b: 2, c: 3 };

const clone = Object.assign({}, existing);
```

This includes `Symbol` properties in addition to `String` ones.

Object rest/spread destructuring which is currently a stage 3 proposal provides an even simpler way to create shallow clones of Object instances:

```
const existing = { a: 1, b: 2, c: 3 };

const { ...clone } = existing;
```

If you need to support older versions of JavaScript, the most-compatible way to clone an Object is by manually iterating over its properties and filtering out inherited ones using `.hasOwnProperty()`.

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
  if (existing.hasOwnProperty(prop)) {
    clone[prop] = existing[prop];
  }
}
```

Section 13.2: Object.freeze

Version ≥ 5

`Object.freeze` makes an object immutable by preventing the addition of new properties, the removal of existing properties, and the modification of the enumerability, configurability, and writability of existing properties. It also prevents the value of existing properties from being changed. However, it does not work recursively which means that child objects are not automatically frozen and are subject to change.

The operations following the freeze will fail silently unless the code is running in strict mode. If the code is in strict mode, a `TypeError` will be thrown.

```
var obj = {
  foo: 'foo',
  bar: [1, 2, 3],
  baz: {
    foo: 'nested-foo'
  }
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
    writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';
```

Section 13.3: Object cloning

When you want a complete copy of an object (i.e. the object properties and the values inside those properties, etc...), that is called **deep cloning**.

Version ≥ 5.1

If an object can be serialized to JSON, then you can create a deep clone of it with a combination of `JSON.parse` and `JSON.stringify`:

```
var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change
```

Note that `JSON.stringify` will convert `Date` objects to ISO-format string representations, but `JSON.parse` will not convert the string back into a `Date`.

There is no built-in function in JavaScript for creating deep clones, and it is not possible in general to create deep clones for every object for many reasons. For example,

- objects can have non-enumerable and hidden properties which cannot be detected.
- object getters and setters cannot be copied.
- objects can have a cyclic structure.
- function properties can depend on state in a hidden scope.

Assuming that you have a "nice" object whose properties only contain primitive values, dates, arrays, or other "nice" objects, then the following function can be used for making deep clones. It is a recursive function that can detect objects with a cyclic structure and will throw an error in such cases.

```
function deepClone(obj) {
    function clone(obj, traversedObjects) {
        var copy;
        // primitive types
        if(obj === null || typeof obj !== "object") {
            return obj;
        }

        // detect cycles
        for(var i = 0; i < traversedObjects.length; i++) {
            if(traversedObjects[i] === obj) {
                throw new Error("Cannot clone circular object.");
            }
        }

        // dates
        if(obj instanceof Date) {
            copy = new Date();
            copy.setTime(obj.getTime());
            return copy;
        }
        // arrays
        if(obj instanceof Array) {
            copy = [];
            for(var i = 0; i < obj.length; i++) {
                copy.push(clone(obj[i], traversedObjects.concat(obj)));
            }
            return copy;
        }
        // simple objects
        if(obj instanceof Object) {
            copy = {};
            for(var key in obj) {
                if(obj.hasOwnProperty(key)) {
                    copy[key] = clone(obj[key], traversedObjects.concat(obj));
                }
            }
            return copy;
        }
        throw new Error("Not a cloneable object.");
    }

    return clone(obj, []);
}
```

Section 13.4: Object properties iteration

You can access each property that belongs to an object with this loop

```
for (var property in object) {
    // always check if an object has a property
    if (object.hasOwnProperty(property)) {
        // do stuff
    }
}
```

You should include the additional check for `hasOwnProperty` because an object may have properties that are inherited from the object's base class. Not performing this check can raise errors.

Version ≥ 5

You can also use `Object.keys` function which return an Array containing all properties of an object and then you can loop through this array with `Array.map` or `Array.forEach` function.

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
    console.log(key);
});
// outputs: 0, 1, 2
```

Section 13.5: Object.assign

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Use it to assign values to an existing object:

```
var user = {
    firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Or to create a shallow copy of an object:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

Or merge many properties from multiple objects to one:

```
var obj1 = {
    a: 1
};
var obj2 = {
    b: 2
};
var obj3 = {
    c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed
```

Primitives will be wrapped, null and undefined will be ignored:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');

var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Logs: { "0": "a", "1": "b", "2": "c" }
```

> Note, only string wrappers can have own enumerable properties

Use it as reducer: (merges an array to an object)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user})
```

Section 13.6: Object rest/spread (...)

Version > 7

Object spreading is just syntactic sugar for `Object.assign({}, obj1, ..., objn);`

It is done with the `...` operator:

```
let obj = { a: 1 };

let obj2 = { ...obj, b: 2, c: 3 };

console.log(obj2); // { a: 1, b: 2, c: 3 };
```

As `Object.assign` it does **shallow** merging, not deep merging.

```
let obj3 = { ...obj, b: { c: 2 } };

console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTE: This specification is currently in stage 3

Section 13.7: Object.defineProperty

Version ≥ 5

It allows us to define a property in an existing object using a property descriptor.

```
var obj = { };

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Console output

> foo

`Object.defineProperty` can be called with the following options:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

`Object.defineProperties` allows you to define multiple properties at a time.

```
var obj = {};
```

```
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Section 13.8: Accesor properties (get and set)

Version ≥ 5

Treat a property as a combination of two functions, one to get the value from it, and another one to set the value in it.

The **get** property of the property descriptor is a function that will be called to retrieve the value from the property.

The **set** property is also a function, it will be called when the property has been assigned a value, and the new value will be passed as an argument.

You cannot assign a `value` or `writable` to a descriptor that has **get** or **set**

```
var person = { name: "John", surname: "Doe"};
Object.defineProperty(person, 'fullName', {
    get: function () {
        return this.name + " " + this.surname;
    },
    set: function (value) {
        [this.name, this.surname] = value.split(" ");
    }
});

console.log(person.fullName); // -> "John Doe"

person.surname = "Hill";
console.log(person.fullName); // -> "John Hill"

person.fullName = "Mary Jones";
console.log(person.name) // -> "Mary"
```

Section 13.9: Dynamic / variable property names

Sometimes the property name needs to be stored into a variable. In this example, we ask the user what word needs to be looked up, and then provide the result from an object I've named `dictionary`.

```
var dictionary = {
    lettuce: 'a veggie',
    banana: 'a fruit',
    tomato: 'it depends on who you ask',
    apple: 'a fruit',
    Apple: 'Steve Jobs rocks!' // properties are case-sensitive
}

var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]
```

```
alert(word + '\n\n' + definition)
```

Note how we are using [] bracket notation to look at the variable named `word`; if we were to use the traditional `.` notation, then it would take the value literally, hence:

```
console.log(dictionary.word)  // doesn't work because word is taken literally and dictionary has no field named `word`
console.log(dictionary.apple) // it works! because apple is taken literally

console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly typed in one of the words from our dictionary when prompted
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

You could also write literal values with [] notation by replacing the variable `word` with a string `'apple'`. See [Properties with special characters or reserved words] example.

You can also set dynamic properties with the bracket syntax:

```
var property="test";
var obj={
 [property]=1;
};

console.log(obj.test);//1
```

It does the same as:

```
var property="test";
var obj={};
obj[property]=1;
```

Section 13.10: Arrays are Objects

> **Disclaimer:** Creating array-like objects is not recommend. However, it is helpful to understand how they work, especially when working with DOM. This will explain why regular array operations don't work on DOM objects returned from many DOM `document` functions. (i.e. `querySelectorAll`, `form.elements`)

Supposing we created the following object which has some properties you would expect to see in an Array.

```
var anObject = {
    foo: 'bar',
    length: 'interesting',
    '0': 'zero!',
    '1': 'one!'
};
```

Then we'll create an array.

```
var anArray = ['zero.', 'one.'];
```

Now, notice how we can inspect both the object, and the array in the same way.

```
console.log(anArray[0], anObject[0]); // outputs: zero.  zero!
console.log(anArray[1], anObject[1]); // outputs: one.  one!
console.log(anArray.length, anObject.length); // outputs: 2 interesting
```

```
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Since `anArray` is actually an object, just like `anObject`, we can even add custom wordy properties to `anArray`

> **Disclaimer:** Arrays with custom properties are not usually recommended as they can be confusing, but it can be useful in advanced cases where you need the optimized functions of an Array. (i.e. jQuery objects)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

We can even make `anObject` to be an array-like object by adding a `length`.

```
anObject.length = 2;
```

Then you can use the C-style **`for`** loop to iterate over `anObject` just as if it were an Array. See Array Iteration

Note that `anObject` is only an **array-like** object. (also known as a List) It is not a true Array. This is important, because functions like `push` and `forEach` (or any convenience function found in `Array.prototype`) will not work by default on array-like objects.

Many of the DOM `document` functions will return a List (i.e. `querySelectorAll`, `form.elements`) which is similar to the array-like `anObject` we created above. See Converting Array-like Objects to Arrays

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true  true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true  true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true  false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true  false
```

Section 13.11: Object.seal

Version ≥ 5

`Object.seal` prevents the addition or removal of properties from an object. Once an object has been sealed its property descriptors can't be converted to another type. Unlike `Object.freeze` it does allow properties to be edited.

Attempts to do this operations on a sealed object will fail silently

```
var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
    get: function () { return 'newFoo'; }
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
```

```
    writable: false
}); // TypeError

obj.foo = 'newFoo';
obj.foo; // 'foo';
```

In strict mode these operations will throw a `TypeError`

```
(function () {
    'use strict';

    var obj = { foo: 'foo' };

    Object.seal(obj);

    obj.newFoo = 'newFoo'; // TypeError
}());
```

Section 13.12: Convert object's values to array

Given this object:

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!",
};
```

You can convert its values to an array by doing:

```
var array = Object.keys(obj)
    .map(function(key) {
        return obj[key];
    });

console.log(array); // ["hello", "this is", "javascript!"]
```

Section 13.13: Retrieving properties from an object

Characteristics of properties :

Properties that can be retrieved from an *object* could have the following characteristics,

- Enumerable
- Non - Enumerable
- own

While creating the properties using *Object.defineProperty(ies)*, we could set its characteristics except *"own"*. Properties which are available in the direct level not in the *prototype* level (`__proto__`) of an object are called as *own* properties.

And the properties that are added into an object without using `Object.defindProperty(ies)` will don't have its enumerable characteristic. That means it be considered as true.

Purpose of enumerability :

The main purpose of setting enumerable characteristics to a property is to make the particular property's availability when retrieving it from its object, by using different programmatical methods. Those different methods will be discussed deeply below.

Methods of retrieving properties :

Properties from an object could be retrieved by the following methods,

1. `for..in` loop

   This loop is very useful in retrieving enumerable properties from an object. Additionally this loop will retrieve enumerable own properties as well as it will do the same retrieval by traversing through the prototype chain until it sees the prototype as null.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});

for(prop in x){
  props.push(prop);
}

console.log(props); //["a"]
```

2. `Object.keys()` function

   This function was unveiled as a part of ECMAScript 5. It is used to retrieve enumerable own properties from an object. Prior to its release people used to retrieve own properties from an object by combining `for..in` loop and `Object.prototype.hasOwnProperty()` function.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.keys(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.keys(x);
```

```
console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);

console.log(props); //["a"]
```

3. `Object.getOwnProperties()` function

   This function will retrieve both enumerable and non enumerable, own properties from an object. It was also released as a part of ECMAScript 5.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]
```

Miscellaneous :

A technique for retrieving all (own, enumerable, non enumerable, all prototype level) properties from an object is given below,

```
function getAllProperties(obj, props = []){
  return obj == null ? props :
           getAllProperties(Object.getPrototypeOf(obj),
               props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__ : { b : 5, c : 15 }};

//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]
```

And this will be supported by the browsers which supports ECMAScript 5.

Section 13.14: Read-Only property

Version ≥ 5

Using property descriptors we can make a property read only, and any attempt to change its value will fail silently, the value will not be changed and no error will be thrown.

The `writable` property in a property descriptor indicates whether that property can be changed or not.

```
var a  = { };

Object.defineProperty(a, 'foo', { value: 'original', writable: false });

a.foo = 'new';

console.log(a.foo);
```

Console output

> original

Section 13.15: Non enumerable property

Version ≥ 5

We can avoid a property from showing up in `for (... in ...)` loops

The `enumerable` property of the property descriptor tells whether that property will be enumerated while looping through the object's properties.

```
var obj = { };

Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
    console.log(obj[prop]);
}
```

Console output

> show

Section 13.16: Lock property description

Version ≥ 5

A property's descriptor can be locked so no changes can be made to it. It will still be possible to use the property normally, assigning and retrieving the value from it, but any attempt to redefine it will throw an exception.

The `configurable` property of the property descriptor is used to disallow any further changes on the descriptor.

```
var obj = {};

// Define 'foo' as read only and lock it
Object.defineProperty(obj, "foo", {
    value: "original value",
    writable: false,
    configurable: false
});

Object.defineProperty(obj, "foo", {writable: true});
```

This error will be thrown:

> TypeError: Cannot redefine property: foo

And the property will still be read only.

```
obj.foo = "new value";
console.log(foo);
```

Console output

> original value

Section 13.17: Object.getOwnPropertyDescriptor

Get the description of a specific property in an object.

```
var sampleObject = {
    hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}
```

Section 13.18: Descriptors and Named Properties

Properties are members of an object. Each named property is a pair of (name, descriptor). The name is a string that allows access (using the dot notation `object.propertyName` or the square brackets notation `object['propertyName']`). The descriptor is a record of fields defining the bevahiour of the property when it is accessed (what happens to the property and what is the value returned from accessing it). By and large, a property associates a name to a behavior (we can think of the behavior as a black box).

There are two types of named properties:

1. *data property*: the property's name is associated with a value.
2. *accessor property*: the property's name is associated with one or two accessor functions.

Demonstration:

```
obj.propertyName1 = 5; //translates behind the scenes into
                       //either assigning 5 to the value field* if it is a data property
                //or calling the set function with the parameter 5 if accessor property
```

```
//*actually whether an assignment would take place in the case of a data property
//also depends on the presence and value of the writable field - on that later on
```

The property's type is determined by its descriptor's fields, and a property cannot be of both types.

Data descriptors -

- Required fields: `value` or `writable` or both
- Optional fields:`configurable`,`enumerable`

Sample:

```
{
   value: 10,
   writable: true;
}
```

Accessor descriptors -

- Required fields: `get` or `set` or both
- Optional fields: `configurable`, `enumerable`

Sample:

```
{
    get: function () {
        return 10;
    },
    enumerable: true
}
```

meaning of fields and their defaults

`configurable`,`enumerable` and `writable`:

- These keys all default to `false`.
- `configurable` is `true` if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.
- `enumerable` is `true` if and only if this property shows up during enumeration of the properties on the corresponding object.
- `writable` is `true` if and only if the value associated with the property may be changed with an assignment operator.

`get` and `set`:

- These keys default to `undefined`.
- `get` is a function which serves as a getter for the property, or `undefined` if there is no getter. The function return will be used as the value of the property.
- `set` is a function which serves as a setter for the property, or `undefined` if there is no setter. The function will receive as only argument the new value being assigned to the property.

`value`:

- This key defaults to `undefined`.
- The value associated with the property. Can be any valid JavaScript value (number, object, function, etc).

Example:

```
    var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
                                                                    // writable:true,
                                                                    // enumerable:true,
                                                                    // configurable:true})
    Object.defineProperty(obj, 'propertyName2', {get: function() {
                                                    console.log('this will be logged ' +
                                'every time propertyName2 is accessed to get its value');
                                                },
                                            set: function() {
                                                    console.log('and this will be logged ' +
                                'every time propertyName2\'s value is tried to be set')
                      //will be treated like it has enumerable:false, configurable:false
                                                }});
//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property





obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to get
its value
```

Section 13.19: Object.keys

Version ≥ 5

`Object.keys(obj)` returns an array of a given object's keys.

```
var obj = {
    a: "hello",
    b: "this is",
    c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]
```

Section 13.20: Properties with special characters or reserved words

While object property notation is usually written as `myObject.property`, this will only allow characters that are normally found in JavaScript variable names, which is mainly letters, numbers and underscore (_).

If you need special characters, such as space, ☺, or user-provided content, this is possible using [] bracket notation.

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

All-digit properties:

In addition to special characters, property names that are all-digits will require bracket notation. However, in this case the property need not be written as a string.

```
myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'
```

However, leading zeros are not recommended as that is interpreted as Octal notation. (TODO, we should produce and link to an example describing octal, hexadecimal and exponent notation)

See also: [Arrays are Objects] example.

Section 13.21: Creating an Iterable object

Version ≥ 6

```
var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
  // The iterator should return an Iterator object
  return {
    // The Iterator object must implement a method, next()
    next: function () {
      // next must itself return an IteratorResult object
      if (!this.iterated) {
        this.iterated = true;
        // The IteratorResult object has two properties
        return {
          // whether the iteration is complete, and
          done: false,
          // the value of the current iteration
          value: 'One'
        };
      }
      return {
        // When iteration is complete, just the done property is needed
        done: true
      };
    },
    iterated: false
  };
};

for (var c of myIterableObject) {
  console.log(c);
}
```

Console output

> One

Section 13.22: Iterating over Object entries - Object.entries()

Version ≥ 8

The proposed `Object.entries()` method returns an array of key/value pairs for the given object. It does not return an iterator like `Array.prototype.entries()`, but the Array returned by `Object.entries()` can be iterated regardless.

```
const obj = {
    one: 1,
    two: 2,
    three: 3
};

Object.entries(obj);
```

Results in:

```
[
    ["one", 1],
    ["two", 2],
    ["three", 3]
]
```

It is an useful way of iterating over the key/value pairs of an object:

```
for(const [key, value] of Object.entries(obj)) {
    console.log(key); // "one", "two" and "three"
    console.log(value); // 1, 2 and 3
}
```

Section 13.23: Object.values()

Version ≥ 8

The `Object.values()` method returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop (the difference being that a for-in loop enumerates properties in the prototype chain as well).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Note:

For browser support, please refer to this link

Chapter 14: Arithmetic (Math)

Section 14.1: Constants

Constants	Description	Approximate
`Math.E`	Base of natural logarithm *e*	2.718
`Math.LN10`	Natural logarithm of 10	2.302
`Math.LN2`	Natural logarithm of 2	0.693
`Math.LOG10E`	Base 10 logarithm of *e*	0.434
`Math.LOG2E`	Base 2 logarithm of *e*	1.442
`Math.PI`	Pi: the ratio of circle circumference to diameter (π)	3.14
`Math.SQRT1_2`	Square root of 1/2	0.707
`Math.SQRT2`	Square root of 2	1.414
`Number.EPSILON`	Difference between one and the smallest value greater than one representable as a Number	2.2204460492503130808472633361816E-16
`Number.MAX_SAFE_INTEGER`	Largest integer `n` such that `n` and `n + 1` are both exactly representable as a Number	2^53 - 1
`Number.MAX_VALUE`	Largest positive finite value of Number	1.79E+308
`Number.MIN_SAFE_INTEGER`	Smallest integer `n` such that `n` and `n - 1` are both exactly representable as a Number	-(2^53 - 1)
`Number.MIN_VALUE`	Smallest positive value for Number	5E-324
`Number.NEGATIVE_INFINITY`	Value of negative infinity (-∞)	
`Number.POSITIVE_INFINITY`	Value of positive infinity (∞)	
`Infinity`	Value of positive infinity (∞)	

Section 14.2: Remainder / Modulus (%)

The remainder / modulus operator (`%`) returns the remainder after (integer) division.

```
console.log( 42 %  10); //  2
console.log( 42 % -10); //  2
console.log(-42 %  10); // -2
console.log(-42 % -10); // -2
console.log(-40 %  10); // -0
console.log( 40 %  10); //  0
```

This operator returns the remainder left over when one operand is divided by a second operand. When the first operand is a negative value, the return value will always be negative, and vice versa for positive values.

In the example above, `10` can be subtracted four times from `42` before there is not enough left to subtract again without it changing sign. The remainder is thus: `42 - 4 * 10 = 2`.

The remainder operator may be useful for the following problems:

1. Test if an integer is (not) divisible by another number:

```
x % 4 == 0 // true if x is divisible by 4
x % 2 == 0 // true if x is even number
```

```
x % 2 != 0 // true if x is odd number
```

Since `0 === -0`, this also works for `x <= -0`.

2. Implement cyclic increment/decrement of value within `[0, n)` interval.

Suppose that we need to increment integer value from `0` to (but not including) `n`, so the next value after `n-1` become `0`. This can be done by such pseudocode:

```
var n = ...; // given n
var i = 0;
function inc() {
    i = (i + 1) % n;
}
while (true) {
    inc();
    // update something with i
}
```

Now generalize the above problem and suppose that we need to allow to both increment and decrement that value from `0` to (not including) `n`, so the next value after `n-1` become `0` and the previous value before `0` become `n-1`.

```
var n = ...; // given n
var i = 0;
function delta(d) { // d - any signed integer
    i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Now we can call `delta()` function passing any integer, both positive and negative, as delta parameter.

Using modulus to obtain the fractional part of a number

```
var myNum = 10 / 4;       // 2.5
var fraction = myNum % 1; // 0.5
myNum = -20 / 7;          // -2.857142857142857
fraction = myNum % 1;     // -0.857142857142857
```

Section 14.3: Rounding

Rounding

`Math.round()` will round the value to the closest integer using *half round up* to break ties.

```
var a = Math.round(2.3);       // a is now 2
var b = Math.round(2.7);       // b is now 3
var c = Math.round(2.5);       // c is now 3
```

But

```
var c = Math.round(-2.7);       // c is now -3
var c = Math.round(-2.5);       // c is now -2
```

Note how `-2.5` is rounded to `-2`. This is because half-way values are always rounded up, that is they're rounded to the integer with the next higher value.

Rounding up

`Math.ceil()` will round the value up.

```
var a = Math.ceil(2.3);        // a is now 3
var b = Math.ceil(2.7);        // b is now 3
```

`ceil`ing a negative number will round towards zero

```
var c = Math.ceil(-1.1);       // c is now 1
```

Rounding down

`Math.floor()` will round the value down.

```
var a = Math.floor(2.3);        // a is now 2
var b = Math.floor(2.7);        // b is now 2
```

`floor`ing a negative number will round it away from zero.

```
var c = Math.floor(-1.1);       // c is now -1
```

Truncating

Caveat: using bitwise operators (except `>>>`) only applies to numbers between `-2147483649` and `2147483648`.

```
2.3  | 0;                       // 2 (floor)
-2.3 | 0;                       // -2 (ceil)
NaN  | 0;                       // 0
```

Version ≥ 6

`Math.trunc()`

```
Math.trunc(2.3);                // 2 (floor)
Math.trunc(-2.3);               // -2 (ceil)
Math.trunc(2147483648.1);       // 2147483648 (floor)
Math.trunc(-2147483649.1);      // -2147483649 (ceil)
Math.trunc(NaN);                // NaN
```

Rounding to decimal places

`Math.floor`, `Math.ceil()`, and `Math.round()` can be used to round to a number of decimal places

To round to 2 decimal places:

```
 var myNum = 2/3;               // 0.6666666666666666
 var multiplier = 100;
 var a = Math.round(myNum * multiplier) / multiplier;  // 0.67
 var b = Math.ceil (myNum * multiplier) / multiplier;  // 0.67
 var c = Math.floor(myNum * multiplier) / multiplier;  // 0.66
```

You can also round to a number of digits:

```
 var myNum = 10000/3;           // 3333.3333333333335
 var multiplier = 1/100;
 var a = Math.round(myNum * multiplier) / multiplier;  // 3300
```

```
var b = Math.ceil (myNum * multiplier) / multiplier;  // 3400
var c = Math.floor(myNum * multiplier) / multiplier;  // 3300
```

As a more usable function:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;    // 3333.3333333333335
roundTo(myNum, 2);  // 3333.33
roundTo(myNum, 0);  // 3333
roundTo(myNum, -2); // 3300
```

And the variants for `ceil` and `floor`:

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

Section 14.4: Trigonometry

All angles below are in radians. An angle `r` in radians has measure `180 * r / Math.PI` in degrees.

Sine

```
Math.sin(r);
```

This will return the sine of `r`, a value between -1 and 1.

```
Math.asin(r);
```

This will return the arcsine (the reverse of the sine) of `r`.

```
Math.asinh(r)
```

This will return the hyperbolic arcsine of `r`.

Cosine

```
Math.cos(r);
```

This will return the cosine of `r`, a value between -1 and 1

```
Math.acos(r);
```

This will return the arccosine (the reverse of the cosine) of `r`.

```
Math.acosh(r);
```

This will return the hyperbolic arccosine of `r`.

Tangent

```
Math.tan(r);
```

This will return the tangent of `r`.

```
Math.atan(r);
```

This will return the arctangent (the reverse of the tangent) of `r`. Note that it will return an angle in radians between `-π/2` and `π/2`.

```
Math.atanh(r);
```

This will return the hyperbolic arctangent of `r`.

```
Math.atan2(x, y);
```

This will return the value of an angle from `(0, 0)` to `(x, y)` in radians. It will return a value between `-π` and `π`, not including π.

Section 14.5: Bitwise operators

Note that all bitwise operations operate on 32-bit integers by passing any operands to the internal function ToInt32.

Bitwise or

```
var a;
a = 0b0011 | 0b1010; // a === 0b1011
// truth table
// 1010 | (or)
// 0011
// 1011  (result)
```

Bitwise and

```
a = 0b0011 & 0b1010; // a === 0b0010
// truth table
// 1010 & (and)
// 0011
// 0010  (result)
```

Bitwise not

```
a = ~0b0011; // a === 0b1100
// truth table
// 10 ~(not)
// 01  (result)
```

Bitwise xor (exclusive or)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// truth table
// 1010 ^ (xor)
// 0011
// 1001  (result)
```

Bitwise left shift

```
a = 0b0001 << 1; // a === 0b0010
a = 0b0001 << 2; // a === 0b0100
a = 0b0001 << 3; // a === 0b1000
```

Shift left is equivalent to integer multiply by `Math.pow(2, n)`. When doing integer math, shift can significantly improve the speed of some math operations.

```
var n = 2;
var a = 5.4;
var result = (a << n) === Math.floor(a) * Math.pow(2,n);
// result is true
a = 5.4 << n; // 20
```

Bitwise right shift `>>` (Sign-propagating shift) `>>>` (Zero-fill right shift)

```
a = 0b1001 >> 1; // a === 0b0100
a = 0b1001 >> 2; // a === 0b0010
a = 0b1001 >> 3; // a === 0b0001

a = 0b1001 >>> 1; // a === 0b0100
a = 0b1001 >>> 2; // a === 0b0010
a = 0b1001 >>> 3; // a === 0b0001
```

A negative 32bit value always has the left most bit on:

```
a = 0b11111111111111111111111111110111 | 0;
console.log(a); // -9
b = a >> 2;     // leftmost bit is shifted 1 to the right then new left most bit is set to on (1)
console.log(b); // -3
b = a >>> 2;    // leftmost bit is shifted 1 to the right. the new left most bit is set to off (0)
console.log(b); // 2147483643
```

The result of a `>>>` operation is always positive.
The result of a `>>` is always the same sign as the shifted value.

Right shift on positive numbers is the equivalent of dividing by the `Math.pow(2,n)` and flooring the result:

```
a = 256.67;
n = 4;
result = (a >> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >> n; //  16

result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is true
a = a >>> n; //  16
```

Right shift zero fill (`>>>`) on negative numbers is different. As JavaScript does not convert to unsigned ints when doing bit operations there is no operational equivalent:

```
a = -256.67;
result = (a >>> n) === Math.floor( Math.floor(a) / Math.pow(2,n) );
// result is false
```

Bitwise assignment operators

With the exception of not (~) all the above bitwise operators can be used as assignment operators:

```
a |= b;   // same as: a = a | b;
a ^= b;   // same as: a = a ^ b;
a &= b;   // same as: a = a & b;
a >>= b;  // same as: a = a >> b;
a >>>= b; // same as: a = a >>> b;
a <<= b;  // same as: a = a << b;
```

Warning: JavaScript uses Big Endian to store integers. This will not always match the Endian of the device/OS. When using typed arrays with bit lengths greater than 8 bits you should check if the environment is Little Endian or Big Endian before applying bitwise operations.

Warning: Bitwise operators such as `&` and `|` are **not** the same as the logical operators `&&` (and) and `||` (or). They will provide incorrect results if used as logical operators. The `^` operator is **not** the power operator (*ab*).

Section 14.6: Incrementing (++)

The Increment operator (`++`) increments its operand by one.

- If used as a postfix, then it returns the value before incrementing.
- If used as a prefix, then it returns the value after incrementing.

```
//postfix
var a = 5,    // 5
    b = a++,  // 5
    c = a     // 6
```

In this case, `a` is incremented after setting `b`. So, `b` will be 5, and `c` will be 6.

```
//prefix
var a = 5,    // 5
    b = ++a,  // 6
    c = a     // 6
```

In this case, `a` is incremented before setting `b`. So, `b` will be 6, and `c` will be 6.

The increment and decrement operators are commonly used in `for` loops, for example:

```
for(var i = 0; i < 42; ++i)
{
  // do something awesome!
}
```

Notice how the *prefix* variant is used. This ensures that a temporarily variable isn't needlessly created (to return the value prior to the operation).

Section 14.7: Exponentiation (Math.pow() or **)

Exponentiation makes the second operand the power of the first operand (ab).

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

`c` will now be 8

`Version > 6`

Stage 3 ES2016 (ECMAScript 7) Proposal:

```
let a = 2,
    b = 3,
```

```
    c = a ** b;
```

`c` will now be 8

Use Math.pow to find the nth root of a number.

Finding the nth roots is the inverse of raising to the nth power. For example `2` to the power of `5` is `32`. The 5th root of `32` is `2`.

```
Math.pow(v, 1 / n); // where v is any positive real number
                    // and n is any positive integer

var a = 16;
var b = Math.pow(a, 1 / 2);  // return the square root of 16 = 4
var c = Math.pow(a, 1 / 3);  // return the cubed root of 16 = 2.5198420997897464
var d = Math.pow(a, 1 / 4);  // return the 4th root of 16 = 2
```

Section 14.8: Random Integers and Floats

```
var a = Math.random();
```

Sample value of `a`: `0.21322848065742162`

`Math.random()` returns a random number between 0 (inclusive) and 1 (exclusive)

```
function getRandom() {
    return Math.random();
}
```

To use `Math.random()` to get a number from an arbitrary range (not `[0,1)`) use this function to get a random number between min (inclusive) and max (exclusive): interval of `[min, max)`

```
function getRandomArbitrary(min, max) {
  return Math.random() * (max - min) + min;
}
```

To use `Math.random()` to get an integer from an arbitrary range (not `[0,1)`) use this function to get a random number between min (inclusive) and max (exclusive): interval of `[min, max)`

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

To use `Math.random()` to get an integer from an arbitrary range (not `[0,1)`) use this function to get a random number between min (inclusive) and max (inclusive): interval of `[min, max]`

```
function getRandomIntInclusive(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Functions taken from
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Section 14.9: Addition (+)

The addition operator (+) adds numbers.

```
var a = 9,
    b = 3,
    c = a + b;
```

c will now be 12

This operand can also be used multiple times in a single assignment:

```
var a = 9,
    b = 3,
    c = 8,
    d = a + b + c;
```

d will now be 20.

Both operands are converted to primitive types. Then, if either one is a string, they're both converted to strings and concatenated. Otherwise, they're both converted to numbers and added.

```
null + null;      // 0
null + undefined; // NaN
null + {};        // "null[object Object]"
null + '';        // "null"
```

If the operands are a string and a number, the number is converted to a string and then they're concatenated, which may lead to unexpected results when working with strings that look numeric.

```
"123" + 1;        // "1231" (not 124)
```

If a boolean value is given in place of any of the number values, the boolean value is converted to a number (`0` for `false`, 1 for `true`) before the sum is calculated:

```
true + 1;         // 2
false + 5;        // 5
null + 1;         // 1
undefined + 1;    // NaN
```

If a boolean value is given alongside a string value, the boolean value is converted to a string instead:

```
true + "1";        // "true1"
false + "bar";     // "falsebar"
```

Section 14.10: Little / Big endian for typed arrays when using bitwise operators

To detect the endian of the device

```
var isLittleEndian = true;
(()=>{
    var buf = new ArrayBuffer(4);
    var buf8 = new Uint8ClampedArray(buf);
```

```
    var data = new Uint32Array(buf);
    data[0] = 0x0F000000;
    if(buf8[0] === 0x0f){
        isLittleEndian = false;
    }
})();
```

Little-Endian stores most significant bytes from right to left.

Big-Endian stores most significant bytes from left to right.

```
var myNum = 0x11223344 | 0;  // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array
```

If the system uses Little-Endian, then the 8bit byte values will be

```
console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11
```

If the system uses Big-Endian, then the 8bit byte values will be

```
console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44
```

Example where Endian type is important

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red,Green,Blue,Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){  // Mask all pixels
    buf32[i] &= mask; //Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Section 14.11: Get Random Between Two Numbers

Returns a random integer between `min` and `max`:

```
function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Examples:

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Section 14.12: Simulating events with different probabilities

Sometimes you may only need to simulate an event with two outcomes, maybe with different probabilities, but you may find yourself in a situation that calls for many possible outcomes with different probabilities. Let's imagine you want to simulate an event that has six equally probable outcomes. This is quite simple.

```
function simulateEvent(numEvents) {
    var event = Math.floor(numEvents*Math.random());
    return event;
}

// simulate fair die
console.log("Rolled a "+(simulateEvent(6)+1));  // Rolled a 2
```

However, you may not want equally probable outcomes. Say you had a list of three outcomes represented as an array of probabilities in percents or multiples of likelihood. Such an example might be a weighted die. You could rewrite the previous function to simulate such an event.

```
function simulateEvent(chances) {
    var sum = 0;
    chances.forEach(function(chance) {
        sum+=chance;
    });
    var rand = Math.random();
    var chance = 0;
    for(var i=0; i<chances.length; i++) {
        chance+=chances[i]/sum;
        if(rand<chance) {
            return i;
        }
    }

    // should never be reached unless sum of probabilities is less than 1
    // due to all being zero or some being negative probabilities
    return -1;
}

// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+(simulateEvent([1,1,1,1,1,2])+1));  // Rolled a 1

// using probabilities
console.log("Rolled a "+(simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1));  // Rolled a 6
```

As you probably noticed, these functions return an index, so you could have more descriptive outcomes stored in an array. Here's an example.

```
var rewards = ["gold coin","silver coin","diamond","god sword"];
var likelihoods = [5,9,1,0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]);  // You get a silver coin
```

Section 14.13: Subtraction (-)

The subtraction operator (-) subtracts numbers.

```
var a = 9;
var b = 3;
var c = a - b;
```

`c` will now be 6

If a string or boolean is provided in place of a number value, it gets converted to a number before the difference is calculated (`0` for **`false`**, `1` for **`true`**):

```
"5" - 1;     // 4
7 - "3";     // 4
"5" - true;  // 4
```

If the string value cannot be converted into a Number, the result will be **`NaN`**:

```
"foo" - 1;      // NaN
100 - "bar";    // NaN
```

Section 14.14: Multiplication (*)

The multiplication operator (`*`) perform arithmetic multiplication on numbers (literals or variables).

```
console.log( 3 *  5); //  15
console.log(-3 *  5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); //  15
```

Section 14.15: Getting maximum and minimum

The `Math.max()` function returns the largest of zero or more numbers.

```
Math.max(4, 12);   //  12
Math.max(-1, -15); // -1
```

The `Math.min()` function returns the smallest of zero or more numbers.

```
Math.min(4, 12);   //  4
Math.min(-1, -15); // -15
```

Getting maximum and minimum from an array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max.apply(Math, arr),
    min = Math.min.apply(Math, arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

ECMAScript 6 spread operator, getting the maximum and minimum of an array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max(...arr),
    min = Math.min(...arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Section 14.16: Restrict Number to Min/Max Range

If you need to clamp a number to keep it inside a specific range boundary

```
function clamp(min, max, val) {
    return Math.min(Math.max(min, +val), max);
}

console.log(clamp(-10, 10, "4.30")); // 4.3
console.log(clamp(-10, 10, -8));     // -8
console.log(clamp(-10, 10, 12));     // 10
console.log(clamp(-10, 10, -15));    // -10
```

Use-case example (jsFiddle)

Section 14.17: Ceiling and Floor

`ceil()`

The `ceil()` method rounds a number *upwards* to the nearest integer, and returns the result.

Syntax:

```
Math.ceil(n);
```

Example:

```
console.log(Math.ceil(0.60)); //  1
console.log(Math.ceil(0.40)); //  1
console.log(Math.ceil(5.1));  //  6
console.log(Math.ceil(-5.1)); // -5
console.log(Math.ceil(-5.9)); // -5
```

`floor()`

The `floor()` method rounds a number *downwards* to the nearest integer, and returns the result.

Syntax:

```
Math.floor(n);
```

Example:

```
console.log(Math.ceil(0.60)); //  0
console.log(Math.ceil(0.40)); //  0
console.log(Math.ceil(5.1));  //  5
console.log(Math.ceil(-5.1)); // -6
console.log(Math.ceil(-5.9)); // -6
```

Section 14.18: Getting roots of a number

Square Root

Use `Math.sqrt()` to find the square root of a number

```
Math.sqrt(16)   #=> 4
```

Cube Root

To find the cube root of a number, use the `Math.cbrt()` function

Version ≥ 6

```
Math.cbrt(27)  #=> 3
```

Finding nth-roots

To find the nth-root, use the `Math.pow()` function and pass in a fractional exponent.

```
Math.pow(64, 1/6) #=> 2
```

Section 14.19: Random with gaussian distribution

The `Math.random()` function should give random numbers that have a standard deviation approaching 0. When picking from a deck of card, or simulating a dice roll this is what we want.

But in most situations this is unrealistic. In the real world the randomness tends to gather around an common normal value. If plotted on a graph you get the classical bell curve or gaussian distribution.

To do this with the `Math.random()` function is relatively simple.

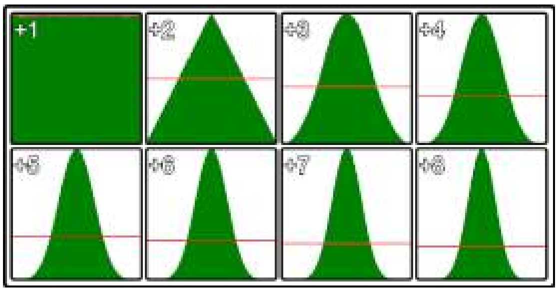
```
var randNum = (Math.random() + Math.random()) / 2;
var randNum = (Math.random() + Math.random() + Math.random()) / 3;
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

Adding a random value to the last increases the variance of the random numbers. Dividing by the number of times you add normalises the result to a range of 0–1

As adding more than a few randoms is messy a simple function will allow you to select a variance you want.

```
// v is the number of times random is summed and should be over >= 1
// return a random number between 0-1 exclusive
function randomG(v){
    var r = 0;
    for(var i = v; i > 0; i --){
        r += Math.random();
```

```
    }
    return r / v;
}
```



The image shows the distribution of random values for different values of v. The top left is standard single `Math.random()` call the bottom right is `Math.random()` summed 8 times. This is from 5,000,000 samples using Chrome

This method is most efficient at `v<5`

Section 14.20: Math.atan2 to find direction

If you are working with vectors or lines you will at some stage want to get the direction of a vector, or line. Or the direction from a point to another point.

`Math.atan(yComponent, xComponent)` return the angle in radius within the range of `-Math.PI` to `Math.PI` (`-180` to `180` deg)

Direction of a vector

```
var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x);  // 0.6435011087932844
```

Direction of a line

```
var line = {
    p1 : { x : 100, y : 128},
    p2 : { x : 320, y : 256}
}
// get the direction from p1 to p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297
```

Direction from a point to another point

```
var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714
```

Section 14.21: Sin & Cos to create a vector given direction & distance

If you have a vector in polar form (direction & distance) you will want to convert it to a cartesian vector with a x and y component. For reference the screen coordinate system has directions as 0 deg points from left to right, 90 (PI/2) point down the screen, and so on in a clock wise direction.

```
var dir = 1.4536; // direction in radians
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component
```

```
vec.y = Math.sin(dir) * dist; // get the y component
```

You can also ignore the distance to create a normalised (1 unit long) vector in the direction of `dir`

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

If your coordinate system has y as up then you need to switch cos and sin. In this case a positive direction is in a counterclockwise direction from the x axis.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Section 14.22: Math.hypot

To find the distance between two points we use pythagoras to get the square root of the sum of the square of the component of the vector between them.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

With ECMAScript 6 came `Math.hypot` which does the same thing

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

Now you don't have to hold the interim vars to stop the code becoming a mess of variables

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

`Math.hypot` can take any number of dimensions

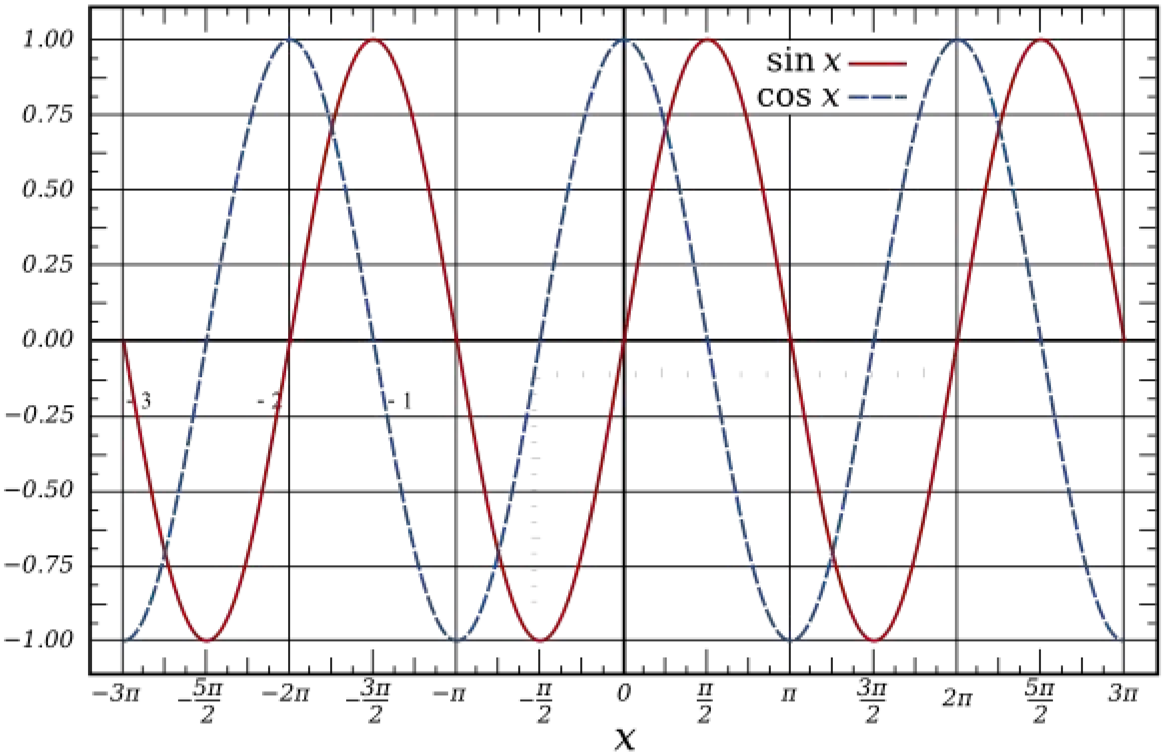
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// find length of 11th dimensional vector
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist = Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

Section 14.23: Periodic functions using Math.sin

`Math.sin` and `Math.cos` are cyclic with a period of 2*PI radians (360 deg) they output a wave with an amplitude of 2 in the range -1 to 1.

Graph of sine and cosine function: *(courtesy Wikipedia)*



They are both very handy for many types of periodic calculations, from creating sound waves, to animations, and even encoding and decoding image data

This example shows how to create a simple sin wave with control over period/frequency, phase, amplitude, and offset.

The unit of time being used is seconds.
The most simple form with control over frequency only.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
}
```

In almost all cases you will want to make some changes to the value returned. The common terms for modifications

- Phase: The offset in terms of frequency from the start of the oscillations. It is a value in the range of 0 to 1 where the value 0.5 move the wave forward in time by half its frequency. A value of 0 or 1 makes no change.
- Amplitude: The distance from the lowest value and highest value during one cycle. An amplitude of 1 has a range of 2. The lowest point (trough) -1 to the highest (peak) 1. For a wave with frequency 1 the peak is at 0.25 seconds, and trough at 0.75.
- Offset: moves the whole wave up or down.

To include all these in the function:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
```

```
    t += phase * Math.PI * 2; // add the phase offset
    var v = Math.sin(t); // get the value at the calculated position in the cycle
    v *= amplitude; // set the amplitude
    v += offset; // add the offset
    return v;
}
```

Or in a more compact (and slightly quicker form):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude + offset;
}
```

All the arguments apart from time are optional

Section 14.24: Division (/)

The division operator (/) perform arithmetic division on numbers (literals or variables).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

Section 14.25: Decrementing (--)

The decrement operator (`--`) decrements numbers by one.

- If used as a postfix to `n`, the operator returns the current `n` and *then* assigns the decremented the value.
- If used as a prefix to `n`, the operator assigns the decremented `n` and *then* returns the changed value.

```
var a = 5,    // 5
    b = a--,  // 5
    c = a     // 4
```

In this case, `b` is set to the initial value of `a`. So, `b` will be 5, and `c` will be 4.

```
var a = 5,    // 5
    b = --a,  // 4
    c = a     // 4
```

In this case, `b` is set to the new value of `a`. So, `b` will be 4, and `c` will be 4.

Common Uses

The decrement and increment operators are commonly used in `for` loops, for example:

```
for (var i = 42; i > 0; --i) {
  console.log(i)
}
```

Notice how the *prefix* variant is used. This ensures that a temporarily variable isn't needlessly created (to return the value prior to the operation).

> **Note:** Neither `--` nor `++` are like normal mathematical operators, but rather they are very concise operators for *assignment*. Notwithstanding the return value, both

> `x--`
>
> and
>
> `--x`
>
> reassign to x such that `x = x - 1`.

```
const x = 1;
console.log(x--)  // TypeError: Assignment to constant variable.
console.log(--x)  // TypeError: Assignment to constant variable.
console.log(--3)  // ReferenceError: Invalid left-hand size expression in prefix operation.
console.log(3--)  // ReferenceError: Invalid left-hand side expression in postfix operation.
```

Chapter 15: Bitwise operators

Section 15.1: Bitwise operators

Bitwise operators perform operations on bit values of data. These operators convert operands to signed 32-bit integers in [two's complement](https://en.wikipedia.org/wiki/Two%27s_complement).

Conversion to 32-bit integers

Numbers with more than 32 bits discard their most significant bits. For example, the following integer with more than 32 bits is converted to a 32-bit integer:

```
Before: 10100110111110100000000010000011110001000001
After:              10100000000010000011110001000001
```

Two's Complement

In normal binary we find the binary value by adding the `1`'s based on their position as powers of 2 - The rightmost bit being `2^0` to the leftmost bit being `2^n-1` where `n` is the number of bits. For example, using 4 bits:

```
// Normal Binary
// 8 4 2 1
   0 1 1 0  => 0 + 4 + 2 + 0 => 6
```

Two complement's format means that the number's negative counterpart (6 vs -6) is all the bits for a number inverted, plus one. The inverted bits of 6 would be:

```
// Normal binary
   0 1 1 0
// One's complement (all bits inverted)
   1 0 0 1  => -8 + 0 + 0 + 1 => -7
// Two's complement (add 1 to one's complement)
   1 0 1 0  => -8 + 0 + 2 + 0 => -6
```

Note: Adding more `1`'s to the left of a binary number does not change its value in two's compliment. The value `1010` and `1111111111010` are both `-6`.

Bitwise AND

The bitwise AND operation `a & b` returns the binary value with a `1` where both binary operands have `1`'s in a specific position, and `0` in all other positions. For example:

```
13 & 7 => 5
// 13:     0..01101
//  7:     0..00111
//-----------------
//  5:     0..00101 (0 + 0 + 4 + 0 + 1)
```

Real world example: Number's Parity Check

Instead of this "masterpiece" (unfortunately too often seen in many real code parts):

```
function isEven(n) {
    return n % 2 == 0;
}
```

```
function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

You can check the (integer) number's parity in much more effective and simple manner:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Bitwise OR

The bitwise OR operation `a | b` returns the binary value with a `1` where either operands or both operands have `1`'s in a specific position, and `0` when both values have `0` in a position. For example:

```
13 | 7 => 15
// 13:     0..01101
//  7:     0..00111
//-----------------
// 15:     0..01111 (0 + 8 + 4 + 2 + 1)
```

Bitwise NOT

The bitwise NOT operation `~a` *flips* the bits of the given value `a`. This means all the `1`'s will become `0`'s and all the `0`'s will become `1`'s.

```
~13 => -14
// 13:     0..01101
//-----------------
//-14:     1..10010 (-16 + 0 + 0 + 2 + 0)
```

Bitwise XOR

The bitwise XOR (*exclusive or*) operation `a ^ b` places a `1` only if the two bits are different. Exclusive or means *either one or the other, but not both*.

```
13 ^ 7 => 10
// 13:     0..01101
//  7:     0..00111
//-----------------
// 10:     0..01010  (0 + 8 + 0 + 2 + 0)
```

Real world example: swapping two integer values without additional memory allocation

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b);// a is now 22 and b is now 11
```

Section 15.2: Shift Operators

Bitwise shifting can be thought as "moving" the bits either left or right, and hence changing the value of the data operated on.

Left Shift

The left shift operator `(value) << (shift amount)` will shift the bits to the left by `(shift amount)` bits; the new bits coming in from the right will be `0`'s:

```
5 << 2 => 20
//  5:      0..000101
// 20:      0..010100 <= adds two 0's to the right
```

Right Shift (*Sign-propagating*)

The right shift operator `(value) >> (shift amount)` is also known as the "Sign-propagating right shift" because it keeps the sign of the initial operand. The right shift operator shifts the `value` the specified `shift amount` of bits to the right. Excess bits shifted off the right are discarded. The new bits coming in from the left will be based on the sign of the initial operand. If the left-most bit was `1` then the new bits will all be `1` and vice-versa for `0`'s.

```
20 >> 2 => 5
// 20:      0..010100
//  5:      0..000101 <= added two 0's from the left and chopped off 00 from the right

-5 >> 3 => -1
// -5:      1..111011
// -2:      1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Right Shift (*Zero fill*)

The zero-fill right shift operator `(value) >>> (shift amount)` will move the bits to the right, and the new bits will be `0`'s. The `0`'s are shifted in from the left, and excess bits to the right are shifted off and discarded. This means it can make negative numbers into positive ones.

```
-30 >>> 2 => 1073741816
//       -30:      111..1100010
//1073741816:      001..1111000
```

Zero-fill right shift and sign-propagating right shift yield the same result for non negative numbers.

Chapter 16: Constructor functions

Section 16.1: Declaring a constructor function

Constructor functions are functions designed to construct a new object. Within a constructor function, the keyword **`this`** refers to a newly created object which values can be assigned to. Constructor functions "return" this new object automatically.

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}
```

Constructor functions are invoked using the **`new`** keyword:

```
let cat = new Cat("Tom");
cat.sound; // Returns "Meow"
```

Constructor functions also have a **`prototype`** property which points to an object whose properties are automatically inherited by all objects created with that constructor:

```
Cat.prototype.speak = function() {
  console.log(this.sound);
}

cat.speak(); // Outputs "Meow" to the console
```

Objects created by constructor functions also have a special property on their prototype called `constructor`, which points to the function used to create them:

```
cat.constructor // Returns the `Cat` function
```

Objects created by constructor functions are also considered to be "instances" of the constructor function by the **`instanceof`** operator:

```
cat instanceof Cat // Returns "true"
```

Chapter 17: Declarations and Assignments

Section 17.1: Modifying constants

Declaring a variable `const` only prevents its value from being *replaced* by a new value. `const` does not put any restrictions on the internal state of an object. The following example shows that a value of a property of a `const` object can be changed, and even new properties can be added, because the object that is assigned to `person` is modified, but not *replaced*.

```
const person = {
    name: "John"
};
console.log('The name of the person is', person.name);

person.name = "Steve";
console.log('The name of the person is', person.name);

person.surname = "Fox";
console.log('The name of the person is', person.name, 'and the surname is', person.surname);
```

Result:

```
The name of the person is John
The name of the person is Steve
The name of the person is Steve and the surname is Fox
```

In this example we've created constant object called `person` and we've reassigned `person.name` property and created new `person.surname` property.

Section 17.2: Declaring and initializing constants

You can initialize a constant by using the `const` keyword.

```
const foo = 100;
const bar = false;
const person = { name: "John" };
const fun = function () = { /* ... */ };
const arrowFun = () => /* ... */ ;
```

Important
You must declare and initialize a constant in the same statement.

Section 17.3: Declaration

There are four principle ways to declare a variable in JavaScript: using the `var`, `let` or `const` keywords, or without a keyword at all ("bare" declaration). The method used determines the resulting scope of the variable, or reassignability in the case of `const`.

- The `var` keyword creates a function-scope variable.
- The `let` keyword creates a block-scope variable.
- The `const` keyword creates a block-scope variable that cannot be reassigned.
- A bare declaration creates a global variable.

```
var a = 'foo';    // Function-scope
```

```
let b = 'foo';    // Block-scope
const c = 'foo';  // Block-scope & immutable reference
```

Keep in mind that you can't declare constants without initializing them at the same time.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(An example of keyword-less variable declaration is not included above for technical reasons. Continue reading to see an example.)

Section 17.4: Undefined

Declared variable without a value will have the value **undefined**

```
var a;

console.log(a); // logs: undefined
```

Trying to retrieve the value of undeclared variables results in a ReferenceError. However, both the type of undeclared and unitialized variables is "undefined":

```
var a;
console.log(typeof a === "undefined"); // logs: true
console.log(typeof variableDoesNotExist === "undefined"); // logs: true
```

Section 17.5: Data Types

JavaScript variables can hold many data types: numbers, strings, arrays, objects and more:

```
// Number
var length = 16;

// String
var message = "Hello, World!";

// Array
var carNames = ['Chevrolet', 'Nissan', 'BMW'];

// Object
var person = {
    firstName: "John",
    lastName: "Doe"
};
```

JavaScript has dynamic types. This means that the same variable can be used as different types:

```
var a;              // a is undefined
var a = 5;          // a is a Number
var a = "John";     // a is a String
```

Section 17.6: Mathematic operations and assignment

Increment by

```
var a = 9,
b = 3;
```

```
b += a;
```

`b` will now be 12

This is functionally the same as

```
b = b + a;
```

Decrement by

```
var a = 9,
b = 3;
b -= a;
```

`b` will now be 6

This is functionally the same as

```
b = b - a;
```

Multiply by

```
var a = 5,
b = 3;
b *= a;
```

`b` will now be 15

This is functionally the same as

```
b = b * a;
```

Divide by

```
var a = 3,
b = 15;
b /= a;
```

`b` will now be 5

This is functionally the same as

```
b = b / a;
```

Version ≥ 7

Raised to the power of

```
var a = 3,
b = 15;
b **= a;
```

`b` will now be 3375

This is functionally the same as

```
b = b ** a;
```

Section 17.7: Assignment

To assign a value to a previously declared variable, use the assignment operator, `=`:

```
a = 6;
b = "Foo";
```

As an alternative to independent declaration and assignment, it is possible to perform both steps in one statement:

```
var a = 6;
let b = "Foo";
```

It is in this syntax that global variables may be declared without a keyword; if one were to declare a bare variable without an assignment immediately afterword, the interpreter would not be able to differentiate global declarations `a;` from references to variables `a;`.

```
c = 5;
c = "Now the value is a String.";
myNewGlobal;    // ReferenceError
```

Note, however, that the above syntax is generally discouraged and is not strict-mode compliant. This is to avoid the scenario in which a programmer inadvertently drops a `let` or `var` keyword from their statement, accidentally creating a variable in the global namespace without realizing it. This can pollute the global namespace and conflict with libraries and the proper functioning of a script. Therefore global variables should be declared and initialized using the `var` keyword in the context of the window object, instead, so that the intent is explicitly stated.

Additionally, variables may be declared several at a time by separating each declaration (and optional value assignment) with a comma. Using this syntax, the var and let keywords need only be used once at the beginning of each statement.

```
globalA = "1", globalB = "2";
let x, y = 5;
var person = 'John Doe',
    foo,
    age = 14,
    date = new Date();
```

Notice in the preceding code snippet that the order in which declaration and assignment expressions occur (`var a, b, c = 2, d;`) does not matter. You may freely intermix the two.

Function declaration effectively creates variables, as well.

Chapter 18: Loops

Section 18.1: Standard "for" loops

Standard usage

```
for (var i = 0; i < 100; i++) {
    console.log(i);
}
```

Expected output:

> 0
> 1
> ...
> 99

Multiple declarations

Commonly used to cache the length of an array.

```
var array = ['a', 'b', 'c'];
for (var i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

Expected output:

> 'a'
> 'b'
> 'c'

Changing the increment

```
for (var i = 0; i < 100; i += 2 /* Can also be: i = i + 2 */) {
    console.log(i);
}
```

Expected output:

> 0
> 2
> 4
> ...
> 98

Decremented loop

```
for (var i = 100; i >=0; i--) {
    console.log(i);
}
```

Expected output:

> 100
> 99
> 98
> ...
> 0

Section 18.2: "for ... of" loop

Version ≥ 6

```
const iterable = [0, 1, 2];
for (let i of iterable) {
    console.log(i);
}
```

Expected output:

> 0
> 1
> 2

The advantages from the for...of loop are:

- This is the most concise, direct syntax yet for looping through array elements
- It avoids all the pitfalls of for...in
- Unlike `forEach()`, it works with break, continue, and return

Support of for...of in other collections
Strings

for...of will treat a string as a sequence of Unicode characters:

```
const string = "abc";
for (let chr of string) {
  console.log(chr);
}
```

Expected output:

> a b c

Sets

for...of works on Set objects.

Note:

- A Set object will eliminate duplicates.
- Please [check this reference](#) for `Set()` browser support.

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];

const uniqueNames = new Set(names);

for (let name of uniqueNames) {
  console.log(name);
}
```

Expected output:

> bob
> alejandro
> zandra
> anna

Maps

You can also use for...of loops to iterate over Maps. This works similarly to arrays and sets, except the iteration variable stores both a key and a value.

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const iteration of map) {
  console.log(iteration) //will log ['abc', 1] and then ['def', 2]
}
```

You can use destructuring assignment to capture the key and the value separately:

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const [key, value] of map) {
  console.log(key + ' is mapped to ' + value)
}
/*Logs:
  abc is mapped to 1
  def is mapped to 2
*/
```

Objects

for...of loops *do not* work directly on plain Objects; but, it is possible to iterate over an object's properties by switching to a for...in loop, or using `Object.keys()`:

```
const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

Expected output:

> name: Mike

Section 18.3: "for ... in" loop

> **Warning**
> for...in is intended for iterating over object keys, not array indexes. [Using it to loop through an array is generally discouraged](). It also includes properties from the prototype, so it may be necessary to check if the key is within the object using `hasOwnProperty`. If any attributes in the object are defined by the `defineProperty/defineProperties` method and set the param `enumerable: false`, those attributes will be inaccessible.

```
var object = {"a":"foo", "b":"bar", "c":"baz"};
// `a` is inaccessible
Object.defineProperty(object , 'a', {
        enumerable: false,
});
for (var key in object) {
    if (object.hasOwnProperty(key)) {
      console.log('object.' + key + ', ' + object[key]);
    }
}
```

Expected output:

> object.b, bar
> object.c, baz

Section 18.4: "while" Loops

Standard While Loop

A standard while loop will execute until the condition given is false:

```
var i = 0;
while (i < 100) {
    console.log(i);
    i++;
}
```

Expected output:

> 0
> 1
> ...
> 99

Decremented loop

```
var i = 100;
while (i > 0) {
```

```
    console.log(i);
    i--; /* equivalent to i=i-1 */
}
```

Expected output:

> 100
> 99
> 98
> ...
> 1

Do...while Loop

A do...while loop will always execute at least once, regardless of whether the condition is true or false:

```
var i = 101;
do {
    console.log(i);
} while (i < 100);
```

Expected output:

> 101

Section 18.5: "continue" a loop

Continuing a "for" Loop

When you put the `continue` keyword in a for loop, execution jumps to the update expression (`i++` in the example):

```
for (var i = 0; i < 3; i++) {
    if (i === 1) {
        continue;
    }
    console.log(i);
}
```

Expected output:

> 0
> 2

Continuing a While Loop

When you `continue` in a while loop, execution jumps to the condition (`i < 3` in the example):

```
var i = 0;
while (i < 3) {
    if (i === 1) {
        i = 2;
        continue;
```

```
    }
    console.log(i);
    i++;
}
```

Expected output:

> 0
> 2

Section 18.6: Break specific nested loops

We can name our loops and break the specific one when necessary.

```
outerloop:
for (var i = 0;i<3;i++){
    innerloop:
    for (var j = 0;j <3; j++){
        console.log(i);
        console.log(j);
        if (j == 1){
            break outerloop;
        }
    }
}
```

Output:

```
0
0
0
1
```

Section 18.7: "do ... while" loop

```
var availableName;
do {
    availableName = getRandomName();
} while (isNameUsed(name));
```

A `do while` loop is guaranteed to run at least once as it's condition is only checked at the end of an iteration. A traditional `while` loop may run zero or more times as its condition is checked at the beginning of an iteration.

Section 18.8: Break and continue labels

Break and continue statements can be followed by an optional label which works like some kind of a goto statement, resumes execution from the label referenced position

```
for(var i = 0; i < 5; i++){
  nextLoop2Iteration:
  for(var j = 0; j < 5; j++){
    if(i == j) break nextLoop2Iteration;
    console.log(i, j);
  }
```

```
}
```

> ***i=0 j=0 skips rest of j values***
> 1 0
> ***i=1 j=1 skips rest of j values***
> 2 0
> 2 1 ***i=2 j=2 skips rest of j values***
> 3 0
> 3 1
> 3 2
> ***i=3 j=3 skips rest of j values***
> 4 0
> 4 1
> 4 2
> 4 3
> ***i=4 j=4 does not log and loops are done***

Chapter 19: Functions

Functions in JavaScript provide organized, reusable code to perform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, return statements and scope in JavaScript.

Section 19.1: Function Scoping

When you define a function, it creates a *scope*.

Everything defined within the function is not accessible by code outside the function. Only code within this scope can see the entities defined inside the scope.

```
function foo() {
  var a = 'hello';
  console.log(a);  // => 'hello'
}

console.log(a);  // reference error
```

Nested functions are possible in JavaScript and the same rules apply.

```
function foo() {
  var a = 'hello';

  function bar() {
    var b = 'world';
    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
  }

  console.log(a);  // => 'hello'
  console.log(b);  // reference error
}

console.log(a);  // reference error
console.log(b);  // reference error
```

When JavaScript tries to resolve a reference or variable, it starts looking for it in the current scope. If it cannot find that declaration in the current scope, it climbs up one scope to look for it. This process repeats until the declaration has been found. If the JavaScript parser reaches the global scope and still cannot find the reference, a reference error will be thrown.

```
var a = 'hello';

function foo() {
  var b = 'world';

  function bar() {
    var c = '!!';

    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
    console.log(c);  // => '!!'
    console.log(d);  // reference error
  }
```

```
}
```

This climbing behavior can also mean that one reference may "shadow" over a similarly named reference in the outer scope since it gets seen first.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a);  // => 'world'
  }
}
```
Version ≥ 6

The way JavaScript resolves scoping also applies to the **const** keyword. Declaring a variable with the **const** keyword implies that you are not allowed to reassign the value, but declaring it in a function will create a new scope and with that a new variable.

```
function foo() {
  const a = true;

  function bar() {
    const a = false;  // different variable
    console.log(a);   // false
  }

  const a = false;    // SyntaxError
  a = false;          // TypeError
  console.log(a);     // true
}
```

However, functions are not the only blocks that create a scope (if you are using **let** or **const**). **let** and **const** declarations have a scope of the nearest block statement. See here for a more detailed description.

Section 19.2: Currying

Currying is the transformation of a function of `n` arity or arguments into a sequence of `n` functions taking only one argument.

Use cases: When the values of some arguments are available before others, you can use currying to decompose a function into a series of functions that complete the work in stages, as each value arrives. This can be useful:

- When the value of an argument almost never changes (e.g., a conversion factor), but you need to maintain the flexibility of setting that value (rather than hard-coding it as a constant).
- When the result of a curried function is useful before the other curried functions have run.
- To validate the arrival of the functions in a specific sequence.

For example, the volume of a rectangular prism can be explained by a function of three factors: length (`l`), width (`w`), and height (`h`):

```
var prism = function(l, w, h) {
    return l * w * h;
}
```

A curried version of this function would look like:

```
function prism(l) {
    return function(w) {
        return function(h) {
            return l * w * h;
        }
    }
}
```

Version ≥ 6

```
// alternatively, with concise ECMAScript 6+ syntax:
var prism = l => w => h => l * w * h;
```

You can call these sequence of functions with `prism(2)(3)(5)`, which should evaluate to 30.

Without some extra machinery (like with libraries), currying is of limited syntactical flexibility in JavaScript (ES 5/6) due to the lack of placeholder values; thus, while you can use `var a = prism(2)(3)` to create a partially applied function, you cannot use `prism()(3)(5)`.

Section 19.3: Immediately Invoked Function Expressions

Sometimes you don't want to have your function accessible/stored as a variable. You can create an Immediately Invoked Function Expression (IIFE for short). These are essentially *self-executing anonymous functions*. They have access to the surrounding scope, but the function itself and any internal variables will be inaccessible from outside. An important thing to note about IIFE is that even if you name your function, IIFE are not hoisted like standard functions are and cannot be called by the function name they are declared with.

```
(function() {
   alert("I've run - but can't be run again because I'm immediately invoked at runtime,
          leaving behind only the result I generate");
}());
```

This is another way to write IIFE. Notice that the closing parenthesis before the semicolon was moved and placed right after the closing curly bracket:

```
(function() {
   alert("This is IIFE too.");
})();
```

You can easily pass parameters into an IIFE:

```
(function(message) {
   alert(message);
}("Hello World!"));
```

Additionally, you can return values to the surrounding scope:

```
var example = (function() {
   return 42;
}());
console.log(example); // => 42
```

If required it is possible to name an IIFE. While less often seen, this pattern has several advantages, such as providing a reference which can be used for a recursion and can make debugging simpler as the name is included in the callstack.

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

While wrapping a function in parenthesis is the most common way to denote to the JavaScript parser to expect an expression, in places where an expression is already expected, the notation can be made more concise:

```
var a = function() { return 42 }();
console.log(a)  // => 42
```

Arrow version of immediately invoked function:

Version ≥ 6

```
(() => console.log("Hello!"))(); // => Hello!
```

Section 19.4: Named Functions

Functions can either be named or unnamed (anonymous functions):

```
var namedSum = function sum (a, b) { // named
    return a + b;
}

var anonSum = function (a, b) { // anonymous
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

> 4
> 4

But their names are private to their own scope:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}

sum(1, 3);
```

> Uncaught ReferenceError: sum is not defined

Named functions differ from the anonymous functions in multiple scenarios:

- When you are debugging, the name of the function will appear in the error/stack trace
- Named functions are hoisted while anonymous functions are not
- Named functions and anonymous functions behave differently when handling recursion
- Depending on ECMAScript version, named and anonymous functions may treat the function `name` property differently

Named functions are hoisted

When using an anonymous function, the function can only be called after the line of declaration, whereas a named function can be called before declaration. Consider

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

> Uncaught TypeError: foo is not a function

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

> bar

Named Functions in a recursive scenario

A recursive function can be defined as:

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

//you could call 'say' directly,
//but this way just illustrates the example
var sayHelloTimes = say;

sayHelloTimes(2);
```

> Hello!
> Hello!

What if somewhere in your code the original function binding gets redefined?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> Hello!
> Uncaught TypeError: say is not a function

This can be solved using a named function

```
// The outer variable can even have the same name as the function
// as they are contained in different scopes
var say = function say (times) {
    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it uses the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> Hello!
> Hello!

And as bonus, the named function can't be set to `undefined`, even from inside:

```
var say = function say (times) {
    // this does nothing
    say = undefined;

    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it's using the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> Hello!
> Hello!

The `name` property of functions

Before ES6, named functions had their `name` properties set to their function names, and anonymous functions had their `name` properties set to the empty string.

Version ≤ 5

```
var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'
```

Post ES6, named and unnamed functions both set their `name` properties:

Version ≥ 6

```
var foo = function () {}
console.log(foo.name); // outputs 'foo'

function foo () {}
console.log(foo.name); // outputs 'foo'

var foo = function bar () {}
console.log(foo.name); // outputs 'bar'
```

Section 19.5: Binding `this` and arguments

Version ≥ 5.1

When you take a reference to a method (a property which is a function) in JavaScript, it usually doesn't remember the object it was originally attached to. If the method needs to refer to that object as **`this`** it won't be able to, and calling it will probably cause a crash.

You can use the `.bind()` method on a function to create a wrapper that includes the value of **`this`** and any number of leading arguments.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so value >
this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

When not in strict mode, a function uses the global object (`window` in the browser) as **`this`**, unless the function is called as a method, bound, or called with the method `.call` syntax.

```
window.x = 12;
```

```
function example() {
  return this.x;
}

console.log(example()); // 12
```

In strict mode **`this`** is **`undefined`** by default

```
window.x = 12;

function example() {
  "use strict";
  return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(...)
```

Version ≥ 7

Bind Operator

The double colon **bind operator** can be used as a shortened syntax for the concept explained above:

```
var log = console.log.bind(console); // long version
const log = ::console.log; // short version

foo.bar.call(foo); // long version
foo::bar(); // short version

foo.bar.call(foo, arg1, arg2, arg3); // long version
foo::bar(arg1, arg2, arg3); // short version

foo.bar.apply(foo, args); // long version
foo::bar(...args); // short version
```

This syntax allows you to write normally, without worrying about binding **`this`** everywhere.

Binding console functions to variables

```
var log = console.log.bind(console);
```

Usage:

```
log('one', '2', 3, [4], {5: 5});
```

Output:

```
one 2 3 [4] Object {5: 5}
```

Why would you do that?

One use case can be when you have custom logger and you want to decide on runtime which one to use.

```
var logger = require('appLogger');

var log = logToServer ? logger.log : console.log.bind(console);
```

Section 19.6: Functions with an Unknown Number of Arguments (variadic functions)

To create a function which accepts an undetermined number of arguments, there are two methods depending on your environment.

Version ≤ 5

Whenever a function is called, it has an Array-like arguments object in its scope, containing all the arguments passed to the function. Indexing into or iterating over this will give access to the arguments, for example

```
function logSomeThings() {
    for (var i = 0; i < arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Note that you can convert `arguments` to an actual Array if need-be; see: Converting Array-like Objects to Arrays

Version ≥ 6

From ES6, the function can be declared with its last parameter using the rest operator (`...`). This creates an Array which holds the arguments from that point onwards

```
function personLogsSomeThings(person, ...msg) {
   msg.forEach(arg => {
       console.log(person, 'says', arg);
   });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Functions can also be called with similar way, the spread syntax

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

This syntax can be used to insert arbitrary number of arguments to any position, and can be used with any iterable(`apply` accepts only array-like objects).

```
const logArguments = (...args) => console.log(args)
function* generateNumbers() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Section 19.7: Anonymous Function

Defining an Anonymous Function

When a function is defined, you often give it a name and then invoke it using that name, like so:

```
foo();

function foo(){
    // ...
}
```

When you define a function this way, the JavaScript runtime stores your function in memory and then creates a reference to that function, using the name you've assigned it. That name is then accessible within the current scope. This can be a very convenient way to create a function, but JavaScript does not require you to assign a name to a function. The following is also perfectly legal:

```
function() {
    // ...
}
```

When a function is defined without a name, it's known as an anonymous function. The function is stored in memory, but the runtime doesn't automatically create a reference to it for you. At first glance, it may appear as if such a thing would have no use, but there are several scenarios where anonymous functions are very convenient.

Assigning an Anonymous Function to a Variable

A very common use of anonymous functions is to assign them to a variable:

```
var foo = function(){ /*...*/ };

foo();
```

This use of anonymous functions is covered in more detail in Functions as a variable

Supplying an Anonymous Function as a Parameter to Another Function

Some functions may accept a reference to a function as a parameter. These are sometimes referred to as "dependency injections" or "callbacks", because it allows the function your calling to "call back" to your code, giving you an opportunity to change the way the called function behaves. For example, the Array object's map function allows you to iterate over each element of an array, then build a new array by applying a transform function to each element.

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

It would be tedious, sloppy and unnecessary to create a named function, which would clutter your scope with a function only needed in this one place and break the natural flow and reading of your code (a colleague would have to leave this code to find your function to understand what's going on).

Returning an Anonymous Function From Another Function

Sometimes it's useful to return a function as the result of another function. For example:

```
var hash = getHashFunction( 'sha1' );
```

```
var hashValue = hash( 'Secret Value' );

function getHashFunction( algorithm ){

    if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
    else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };

}
```

Immediately Invoking an Anonymous Function

Unlike many other languages, scoping in JavaScript is function-level, not block-level. (See Function Scoping). In some cases, however, it's necessary to create a new scope. For example, it's common to create a new scope when adding code via a `<script>` tag, rather than allowing variable names to be defined in the global scope (which runs the risk of other scripts colliding with your variable names). A common method to handle this situation is to define a new anonymous function and then immediately invoke it, safely hiding you variables within the scope of the anonymous function and without making your code accessible to third-parties via a leaked function name. For example:

```
<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There is a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
(function(){
    var foo = '';
}()) // <--- the parentheses invokes the function immediately
</script>
```

Self-Referential Anonymous Functions

Sometimes it's useful for an anonymous function to be able to refer to itself. For example, the function may need to recursively call itself or add properties to itself. If the function is anonymous, though, this can be very difficult as it requires knowledge of the variable that the function has been assigned to. This is the less than ideal solution:

```
var foo = function(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if (callAgain === true) foo(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
```

```
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Bad.
```

The intent here was for the anonymous function to recursively call itself, but when the value of foo changes, you end up with a potentially difficult to trace bug.

Instead, we can give the anonymous function a reference to itself by giving it a private name, like so:

```
var foo = function myself(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Whassup?
```

Note that the function name is scoped to itself. The name has not leaked into the outer scope:

```
myself(false); // ReferenceError: myself is not defined
```

This technique is especially useful when dealing with recursive anonymous functions as callback parameters:

Version ≥ 5

```
// Calculate the Fibonacci value for each number in an array:
var fib = false,
    result = [1,2,3,4,5,6,7,8].map(
        function fib(n){
            return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
        });
// result = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Section 19.8: Default parameters

Before ECMAScript 2015 (ES6), a parameter's default value could be assigned in the following way:

```
function printMsg(msg) {
```

```
    msg = typeof msg !== 'undefined' ? // if a value was provided
          msg :                        // then, use that value in the reassignment
          'Default value for msg.';    // else, assign a default value
    console.log(msg);
}
```

ES6 provided a new syntax where the condition and reassignment depicted above is no longer necessary:

Version ≥ 6

```
function printMsg(msg='Default value for msg.') {
    console.log(msg);
}
```

```
printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg in different!'); // -> "Now my msg in different!"
```

This also shows that if a parameter is missing when the function is invoked, its value is kept as `undefined`, as it can be confirmed by explicitly providing it in the following example (using an arrow function):

Version ≥ 6

```
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "str is of type: string"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"
```

Functions/variables as default values and reusing parameters

The default parameters' values are not restricted to numbers, strings or simple objects. A function can also be set as the default value `callback = function(){}`:

Version ≥ 6

```
function foo(callback = function(){ console.log('default'); }) {
    callback();
}

foo(function (){
    console.log('custom');
});
// custom

foo();
//default
```

There are certain characteristics of the operations that can be performed through default values:

- A previously declared parameter can be reused as a default value for the upcoming parameters' values.
- Inline operations are allowed when assigning a default value to a parameter.
- Variables existing in the same scope of the function being declared can be used in its default values.
- Functions can be invoked in order to provide their return value into a default value.

Version ≥ 6

```
let zero = 0;
function multiply(x) { return x * 2;}

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
    console.log((a + b + c), d);
```

```
}

add(1);                    // 4, 4
add(3);                    // 12, 12
add(2, 7);                 // 18, 18
add(1, 2, 5);              // 8, 10
add(1, 2, 5, 10);          // 8, 20
```

Reusing the function's return value in a new invocation's default value:

Version ≥ 6

```
let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);         // [5]
add(6);         // [6], not [5, 6]
add(6, add(5)); // [5, 6]
```

`arguments` value and length when lacking parameters in invocation

The `arguments` array object only retains the parameters whose values are not default, i.e. those that are explicitly provided when the function is invoked:

Version ≥ 6

```
function foo(a = 1, b = a + 1) {
    console.info(arguments.length, arguments);
    console.log(a,b);
}

foo();        // info: 0 >> []     | log: 1, 2
foo(4);       // info: 1 >> [4]    | log: 4, 5
foo(5, 6);    // info: 2 >> [5, 6] | log: 5, 6
```

Section 19.9: Call and apply

Functions have two built-in methods that allow the programmer to supply arguments and the **`this`** variable differently: `call` and `apply`.

This is useful, because functions that operate on one object (the object that they are a property of) can be repurposed to operate on another, compatible object. Additionally, arguments can be given in one shot as arrays, similar to the spread (`...`) operator in ES6.

```
let obj = {
    a: 1,
    b: 2,
    set: function (a, b) {
        this.a = a;
        this.b = b;
    }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property
```

```
obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array

console.log(myObj); // prints { a: 3, b: 5 }
```

Version ≥ 5

ECMAScript 5 introduced another method called **bind()** in addition to call() and apply() to explicitly set **this** value of the function to specific object.

It behaves quite differently than the other two. The first argument to bind() is the **this** value for the new function. All other arguments represent named parameters that should be permanently set in the new function.

```
function showName(label) {
    console.log(label + ":" + this.name);
}
var student1 = {
     name: "Ravi"
};
var student2 = {
     name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"
```

Section 19.10: Partial Application

Similar to currying, partial application is used to reduce the number of arguments passed to a function. Unlike currying, the number need not go down by one.

Example:

This function ...

```
function multiplyThenAdd(a, b, c) {
    return a * b + c;
}
```

... can be used to create another function that will always multiply by 2 and then add 10 to the passed value;

```
function reversedMultiplyThenAdd(c, b, a) {
    return a * b + c;
}

function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);
```

```
multiplyTwoThenAddTen(10); // 30
```

The "application" part of partial application simply means fixing parameters of a function.

Section 19.11: Passing arguments by reference or value

In JavaScript all arguments are passed by value. When a function assigns a new value to an argument variable, that change will not be visible to the caller:

```
var obj = {a: 2};
function myfunc(arg){
    arg = {a: 5}; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

However, changes made to (nested) properties *of* such arguments, will be visible to the caller:

```
var obj = {a: 2};
function myfunc(arg){
    arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

This can be seen as a *call by reference*: although a function cannot change the caller's object by assigning a new value to it, it could *mutate* the caller's object.

As primitive valued arguments, like numbers or strings, are immutable, there is no way for a function to mutate them:

```
var s = 'say';
function myfunc(arg){
    arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

When a function wants to mutate an object passed as argument, but does not want to actually mutate the caller's object, the argument variable should be reassigned:

Version ≥ 6

```
var obj = {a: 2, b: 3};
function myfunc(arg){
    arg = Object.assign({}, arg); // assignment to argument variable, shallow copy
    arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

As an alternative to in-place mutation of an argument, functions can create a new value, based on the argument, and return it. The caller can then assign it, even to the original variable that was passed as argument:

```
var a = 2;
function myfunc(arg){
    arg++;
    return arg;
```

```
}
a = myfunc(a);
console.log(obj.a); // 3
```

Section 19.12: Function Arguments, "arguments" object, rest and spread parameters

Functions can take inputs in form of variables that can be used and assigned inside their own scope. The following function takes two numeric values and returns their sum:

```
function addition (argument1, argument2){
    return argument1 + argument2;
}

console.log(addition(2, 3)); // -> 5
```

`arguments` object

The `arguments` object contains all the function's parameters that contain a non-default value. It can also be used even if the parameters are not explicitly declared:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Although when printing `arguments` the output resembles an Array, it is in fact an object:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Rest parameters: `function (...parm) {}`

In ES6, the `...` syntax when used in the declaration of a function's parameters transforms the variable to its right into a single object containing all the remaining parameters provided after the declared ones. This allows the function to be invoked with an unlimited number of arguments, which will become part of this variable:

```
(function(a, ...b){console.log(typeof b+': '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Spread parameters: `function_name(...varb);`

In ES6, the `...` syntax can also be used when invoking a function by placing an object/variable to its right. This allows that object's elements to be passed into that function as a single object:

```
let nums = [2,42,-1];
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Section 19.13: Function Composition

Composing multiple functions into one is a functional programming common practice;

composition makes a pipeline through which our data will transit and get modified simply working on the function-composition (just like snapping pieces of a track together)...

you start out with some single responsibility functions:

Version ≥ 6

```
 const capitalize = x => x.replace(/^\w/, m => m.toUpperCase());
 const sign = x => x + ',\nmade with love';
```

and easily create a transformation track:

Version ≥ 6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

N.B. Composition is achieved through a utility function usually called `compose` as in our example.

Implementation of `compose` are present in many JavaScript utility libraries (lodash, rambda, etc.) but you can also start out with a simple implementation such as:

Version ≥ 6

```
const compose = (...funs) =>
  x =>
  funs.reduce((ac, f) => f(ac), x);
```

Section 19.14: Get the name of a function object

Version ≥ 6

ES6:

```
myFunction.name
```

Explanation on MDN. As of 2015 works in Node.js and all major browsers except IE.

Version ≥ 5

ES5:

If you have a reference to the function, you can do:

```
function functionName( func )
{
    // Match:
    // - ^          the beginning of the string
    // - function   the word 'function'
    // - \s+        at least some white space
    // - ([\w\$]+)  capture one or more valid JavaScript identifier characters
    // - \(         followed by an opening brace
    //
    var result = /^function\s+([\w\$]+)\(/.exec( func.toString() )

    return result ? result[1] : ''
}
```

Section 19.15: Recursive Function

A recursive function is simply a function, that would call itself.

```
function factorial (n) {
    if (n <= 1) {
        return 1;
    }
```

```
    return n * factorial(n - 1);
}
```

The above function shows a basic example of how to perform a recursive function to return a factorial.

Another example, would be to retrieve the sum of even numbers in an array.

```
function countEvenNumbers (arr) {
    // Sentinel value.  Recursion stops on empty array.
    if (arr.length < 1) {
        return 0;
    }
    // The shift() method removes the first element from an array
    // and returns that element. This method changes the length of the array.
    var value = arr.shift();

    // `value % 2 === 0` tests if the number is even or odd
    // If it's even we add one to the result of counting the remainder of
    // the array.  If it's odd, we add zero to it.
    return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);
}
```

It is important that such functions make some sort of sentinel value check to avoid infinite loops. In the first example above, when `n` is less than or equal to 1, the recursion stops, allowing the result of each call to be returned back up the call stack.

Section 19.16: Using the Return Statement

The return statement can be a useful way to create output for a function. The return statement is especially useful if you do not know in which context the function will be used yet.

```
//An example function that will take a string as input and return
//the first character of the string.

function firstChar (stringIn){
    return stringIn.charAt(0);
}
```

Now to use this function, you need to put it in place of a variable somewhere else in your code:

Using the function result as an argument for another function:

```
console.log(firstChar("Hello world"));
```

Console output will be:

```
> H
```

The return statement ends the function

If we modify the function in the beginning, we can demonstrate that the return statement ends the function.

```
function firstChar (stringIn){
    console.log("The first action of the first char function");
    return stringIn.charAt(0);
    console.log("The last action of the first char function");
```

```
}
```

Running this function like so will look like this:

```
console.log(firstChar("JS"));
```

Console output:

```
> The first action of the first char function
> J
```

It will not print the message after the return statement, as the function has now been ended.

Return statement spanning multiple lines:

In JavaScript, you can normally split up a line of code into many lines for readability purposes or organization. This is valid JavaScript:

```
var
    name = "bob",
    age = 18;
```

When JavaScript sees an incomplete statement like `var` it looks to the next line to complete itself. However, if you make the same mistake with the `return` statement, you will not get what you expected.

```
return
    "Hi, my name is "+ name + ". " +
    "I'm "+ age + " years old.";
```

This code will return `undefined` because `return` by itself is a complete statement in JavaScript, so it will not look to the next line to complete itself. If you need to split up a `return` statement into multiple lines, put a value next to return before you split it up, like so.

```
return "Hi, my name is " + name + ". " +
    "I'm " + age + " years old.";
```

Section 19.17: Functions as a variable

A normal function declaration looks like this:

```
function foo(){
}
```

A function defined like this is accessible from anywhere within its context by its name. But sometimes it can be useful to treat function references like object references. For example, you can assign an object to a variable based on some set of conditions and then later retrieve a property from one or the other object:

```
var name = 'Cameron';
var spouse;

if ( name === 'Taylor' ) spouse = { name: 'Jordan' };
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };

var spouseName = spouse.name;
```

In JavaScript, you can do the same thing with functions:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };

hash('Fred');
```

In the example above, `hash` is a normal variable. It is assigned a reference to a function, after which the function it references can be invoked using parentheses, just like a normal function declaration.

The example above references anonymous functions... functions that do not have their own name. You can also use variables to refer to named functions. The example above could be rewritten like so:

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = sha1Hash;
else if ( hashAlgorithm === 'md5' ) hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}

function sha1Hash(value){
    // ...
}
```

Or, you can assign function references from object properties:

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /**/ },
    md5: function(value) { /**/ }
};

var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = hashAlgorithms.sha1;
else if ( hashAlgorithm === 'md5' ) hash = hashAlgorithms.md5;

hash('Fred');
```

You can assign the reference to a function held by one variable to another by omitting the parentheses. This can result in an easy-to-make mistake: attempting to assign the return value of a function to another variable, but accidentally assigning the reference to the function.

```
// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)
```

```
function getValue(){
    return 41;
}
```

A reference to a function is like any other value. As you've seen, a reference can be assigned to a variable, and that variable's reference value can be subsequently assigned to other variables. You can pass around references to functions like any other value, including passing a reference to a function as the return value of another function. For example:

```
// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');


// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
    // return a reference to an anonymous function
    if (algorithmName === 'sha1') return function(value){ /**/ };
    // return a reference to a declared function
    else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
    // ...
}
```

You don't need to assign a function reference to a variable in order to invoke it. This example, building off example 5, will call getHashingFunction and then immediately invoke the returned function and pass its return value to hashedValue.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

A Note on Hoisting

Keep in mind that, unlike normal function declarations, variables that reference functions are not "hoisted". In example 2, the `md5Hash` and `sha1Hash` functions are defined at the bottom of the script, but are available everywhere immediately. No matter where you define a function, the interpreter "hoists" it to the top of its scope, making it immediately available. This is **not** the case for variable definitions, so code like the following will break:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```

Chapter 20: Functional JavaScript

Section 20.1: Higher-Order Functions

In general, functions that operate on other functions, either by taking them as arguments or by returning them (or both), are called higher-order functions.

A higher-order function is a function that can take another function as an argument. You are using higher-order functions when passing callbacks.

```
function iAmCallbackFunction() {
    console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
    // do some stuff ...

    // invoke the callback function.
    callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

A higher-order function is also a function that returns another function as its result.

```
function iAmJustFunction() {
    // do some stuff ...

    // return a function.
    return function iAmReturnedFunction() {
        console.log("returned function has been invoked");
    }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Section 20.2: Identity Monad

This is an example of an implementation of the identity monad in JavaScript, and could serve as a starting point to create other monads.

Based on the [conference by Douglas Crockford on monads and gonads](#)

Using this approach reusing your functions will be easier because of the flexibility this monad provides, and composition nightmares:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

readable, nice and clean:

```
identityMonad(value)
    .bind(k)
    .bind(j, j1, j2)
    .bind(i, i2)
```

```
    .bind(h, h1, h2)
    .bind(g, g1, g2)
    .bind(f, f1, f2);
```

```
function identityMonad(value) {
    var monad = Object.create(null);

    // func should return a monad
    monad.bind = function (func, ...args) {
        return func(value, ...args);
    };

    // whatever func does, we get our monad back
    monad.call = function (func, ...args) {
        func(value, ...args);

        return identityMonad(value);
    };

    // func doesn't have to know anything about monads
    monad.apply = function (func, ...args) {
        return identityMonad(func(value, ...args));
    };

    // Get the value wrapped in this monad
    monad.value = function () {
        return value;
    };

    return monad;
};
```

It works with primitive values

```
var value = 'foo',
    f = x => x + ' changed',
    g = x => x + ' again';

identityMonad(value)
    .apply(f)
    .apply(g)
    .bind(alert); // Alerts 'foo changed again'
```

And also with objects

```
var value = { foo: 'foo' },
    f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
    g = x => Object.assign(x, { bar: 'foo' }),
    h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
    .bind(f)
    .apply(g)
    .bind(h); // Logs 'foo: bar, bar: foo'
```

Let's try everything:

```
var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
    multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
```

```
    divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
    log = x => console.log(x),
    substract = (x, ...args) => x - add(...args);

identityMonad(100)
    .apply(add, 10, 29, 13)
    .apply(multiply, 2)
    .bind(divideMonad, 2)
    .apply(substract, 67, 34)
    .apply(multiply, 1239)
    .bind(divideMonad, 20, 54, 2)
    .apply(Math.round)
    .call(log); // Logs 29
```

Section 20.3: Pure Functions

A basic principle of functional programming is that it **avoids changing** the application state (statelessness) and variables outside its scope (immutability).

Pure functions are functions that:

- with a given input, always return the same output
- they do not rely on any variable outside their scope
- they do not modify the state of the application (**no side effects**)

Let's take a look at some examples:

> Pure functions must not change any variable outside their scope

Impure function

```
let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1
```

The function changed the `obj.a` value that is outside its scope.

Pure function

```
let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;
  return output;
}
```

```
let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

The function did not change the object `obj` values

> Pure functions must not rely on variables outside their scope

Impure function

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

This **impure** function rely on variable `a` that is defined outside its scope. So, if a is modified, `impure`'s function result will be different.

Pure function

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

The `pure`'s function result **does not rely** on any variable outside its scope.

Section 20.4: Accepting Functions as Arguments

```
function transform(fn, arr) {
    let result = [];
    for (let el of arr) {
        result.push(fn(el)); // We push the result of the transformed item to result
    }
    return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

As you can see, our `transform` function accepts two parameters, a function and a collection. It will then iterate the collection, and push values onto the result, calling `fn` on each of them.

Looks familiar? This is very similar to how `Array.prototype.map()` works!

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Chapter 21: Prototypes, objects

In the conventional JS there are no class instead we have prototypes. Like the class, prototype inherits the properties including the methods and the variables declared in the class. We can create the new instance of the object whenever it is necessary by, Object.create(PrototypeName); (we can give the value for the constructor as well)

Section 21.1: Creation and initialising Prototype

```
var Human = function() {
  this.canWalk = true;
  this.canSpeak = true; //

};

Person.prototype.greet = function() {
  if (this.canSpeak) { // checks whether this prototype has instance of speak
    this.name = "Steve"
    console.log('Hi, I am ' + this.name);
  } else{
     console.log('Sorry i can not speak');
  }
};
```

The prototype can be instantiated like this

```
obj = Object.create(Person.prototype);
ob.greet();
```

We can pass value for the constructor and make the boolean true and false based on the requirement.

Detailed Explanation

```
var Human = function() {
    this.canSpeak = true;
};
// Basic greet function which will greet based on the canSpeak flag
Human.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name);
    }
};

var Student = function(name, title) {
    Human.call(this); // Instantiating the Human object and getting the memebers of the class
    this.name = name; // inheriting the name from the human class
    this.title = title; // getting the title from the called function
};

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;

Student.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name + ', the ' + this.title);
    }
};
```

```
var Customer = function(name) {
    Human.call(this); // inheriting from the base class
    this.name = name;
};

Customer.prototype = Object.create(Human.prototype); // creating the object
Customer.prototype.constructor = Customer;


var bill = new Student('Billy', 'Teacher');
var carter = new Customer('Carter');
var andy = new Student('Andy', 'Bill');
var virat = new Customer('Virat');

bill.greet();
// Hi, I am Bob, the Teacher

carter.greet();
// Hi, I am Carter

andy.greet();
// Hi, I am Andy, the Bill

virat.greet();
```

Chapter 22: Classes

Section 22.1: Class Constructor

The fundamental part of most classes is its constructor, which sets up each instance's initial state and handles any parameters that were passed when calling **new**.

It's defined in a class block as though you're defining a method named constructor, though it's actually handled as a special case.

```
class MyClass {
    constructor(option) {
        console.log(`Creating instance using ${option} option.`);
        this.option = option;
    }
}
```

Example usage:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

A small thing to note is that a class constructor cannot be made static via the **static** keyword, as described below for other methods.

Section 22.2: Class Inheritance

Inheritance works just like it does in other object-oriented languages: methods defined on the superclass are accessible in the extending subclass.

If the subclass declares its own constructor then it must invoke the parents constructor via super() before it can access **this**.

```
class SuperClass {

    constructor() {
        this.logger = console.log;
    }

    log() {
        this.logger(`Hello ${this.name}`);
    }

}

class SubClass extends SuperClass {

    constructor() {
        super();
        this.name = 'subclass';
    }

}

const subClass = new SubClass();

subClass.log(); // logs: "Hello subclass"
```

Section 22.3: Static Methods

Static methods and properties are defined on *the class/constructor itself*, not on instance objects. These are specified in a class definition by using the **static** keyword.

```
class MyClass {
    static myStaticMethod() {
        return 'Hello';
    }

    static get myStaticProperty() {
        return 'Goodbye';
    }
}

console.log(MyClass.myStaticMethod()); // logs: "Hello"
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

We can see that static properties are not defined on object instances:

```
const myClassInstance = new MyClass();

console.log(myClassInstance.myStaticProperty); // logs: undefined
```

However, they *are* defined on subclasses:

```
class MySubClass extends MyClass {};

console.log(MySubClass.myStaticMethod()); // logs: "Hello"
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Section 22.4: Getters and Setters

Getters and setters allow you to define custom behaviour for reading and writing a given property on your class. To the user, they appear the same as any typical property. However, internally a custom function you provide is used to determine the value when the property is accessed (the getter), and to perform any necessary changes when the property is assigned (the setter).

In a `class` definition, a getter is written like a no-argument method prefixed by the **get** keyword. A setter is similar, except that it accepts one argument (the new value being assigned) and the **set** keyword is used instead.

Here's an example class which provides a getter and setter for its `.name` property. Each time it's assigned, we'll record the new name in an internal `.names_` array. Each time it's accessed, we'll return the latest name.

```
class MyClass {
    constructor() {
        this.names_ = [];
    }

    set name(value) {
        this.names_.push(value);
    }

    get name() {
        return this.names_[this.names_.length - 1];
    }
}
```

```
const myClassInstance = new MyClass();
myClassInstance.name = 'Joe';
myClassInstance.name = 'Bob';

console.log(myClassInstance.name); // logs: "Bob"
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]
```

If you only define a setter, attempting to access the property will always return `undefined`.

```
const classInstance = new class {
    set prop(value) {
        console.log('setting', value);
    }
};

classInstance.prop = 10; // logs: "setting", 10

console.log(classInstance.prop); // logs: undefined
```

If you only define a getter, attempting to assign the property will have no effect.

```
const classInstance = new class {
    get prop() {
        return 5;
    }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Section 22.5: Private Members

JavaScript does not technically support private members as a language feature. Privacy - [described by Douglas Crockford](#) - gets emulated instead via closures (preserved function scope) that will be generated each with every instantiation call of a constructor function.

The `Queue` example demonstrates how, with constructor functions, local state can be preserved and made accessible too via privileged methods.

```
class Queue {

  constructor () {                    // - does generate a closure with each instantiation.

    const list = [];                  // - local state ("private member").

    this.enqueue = function (type) {  // - privileged public method
                                      //   accessing the local state
      list.push(type);                //   "writing" alike.
      return type;
    };
    this.dequeue = function () {      // - privileged public method
                                      //   accessing the local state
      return list.shift();            //   "reading / writing" alike.
    };
  }
}
```

```
var q = new Queue;              //
                                //
q.enqueue(9);                   // ... first in ...
q.enqueue(8);                   //
q.enqueue(7);                   //
                                //
console.log(q.dequeue());       // 9 ... first out.
console.log(q.dequeue());       // 8
console.log(q.dequeue());       // 7
console.log(q);                 // {}
console.log(Object.keys(q));    // ["enqueue","dequeue"]
```

With every instantiation of a `Queue` type the constructor generates a closure.

Thus both of a `Queue` type's own methods `enqueue` and `dequeue` (see `Object.keys(q)`) still do have access to `list` that continues to *live* in its enclosing scope that, at construction time, has been preserved.

Making use of this pattern - emulating private members via privileged public methods - one should keep in mind that, with every instance, additional memory will be consumed for every *own property* method (for it is code that can't be shared/reused). The same is true for the amount/size of state that is going to be stored within such a closure.

Section 22.6: Methods

Methods can be defined in classes to perform a function and optionally return a result.
They can receive arguments from the caller.

```
class Something {
    constructor(data) {
        this.data = data
    }

    doSomething(text) {
        return {
            data: this.data,
            text
        }
    }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }
```

Section 22.7: Dynamic Method Names

There is also the ability to evaluate expressions when naming methods similar to how you can access an objects' properties with []. This can be useful for having dynamic property names, however is often used in conjunction with Symbols.

```
let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  // example using symbols
```

```
  [METADATA]() {
    return {
      make: this.make,
      model: this.model
    };
  }

  // you can also use any javascript expression

  // this one is just a string, and could also be defined with simply add()
  ["add"](a, b) {
    return a + b;
  }

  // this one is dynamically evaluated
  [1 + 2]() {
    return "three";
  }
}

let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }
```

Section 22.8: Managing Private Data with Classes

One of the most common obstacles using classes is finding the proper approach to handle private states. There are 4 common solutions for handling private states:

Using Symbols

Symbols are new primitive type introduced on in ES2015, as defined at MDN

> A symbol is a unique and immutable data type that may be used as an identifier for object properties.

When using symbol as a property key, it is not enumerable.

As such, they won't be revealed using `for var in` or `Object.keys`.

Thus we can use symbols to store private data.

```
const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the scope of
the module file
export class SecretAgent{
    constructor(secret){
        this[topSecret] = secret; // we have access to the symbol key (closure)
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret[topSecret]); // we have access to topSecret
        };
    }
}
```

Because `symbols` are unique, we must have reference to the original symbol to access the private property.

```
import {SecretAgent} from 'SecretAgent.js'
```

```
const agent = new SecretAgent('steal all the ice cream');
// ok let's try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the
original symbol will help us to get the data.
```

But it's not 100% private; let's break that agent down! We can use the `Object.getOwnPropertySymbols` method to get the object symbols.

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

Using WeakMaps

`WeakMap` is a new type of object that have been added for es6.

As defined on [MDN](MDN)

> The WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. The keys must be objects and the values can be arbitrary values.

Another important feature of `WeakMap` is, as defined on [MDN](MDN).

> The key in a WeakMap is held weakly. What this means is that, if there are no other strong references to the key, the entire entry will be removed from the WeakMap by the garbage collector.

The idea is to use the WeakMap, as a static map for the whole class, to hold each instance as key and keep the private data as a value for that instance key.

Thus only inside the class will we have access to the `WeakMap` collection.

Let's give our agent a try, with `WeakMap`:

```
const topSecret = new WeakMap(); // will hold all private data of all instances.
export class SecretAgent{
    constructor(secret){
        topSecret.set(this,secret); // we use this, as the key, to set it on our instance private
data
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(topSecret.get(this)); // we have access to topSecret
        };
    }
}
```

Because the const `topSecret` is defined inside our module closure, and since we didn't bind it to our instance properties, this approach is totally private, and we can't reach the agent `topSecret`.

Define all methods inside the constructor

The idea here is simply to define all our methods and members inside the constructor and use the closure to access private members without assigning them to `this`.

```
   export class SecretAgent{
```

```
        constructor(secret){
            const topSecret = secret;
            this.coverStory = 'just a simple gardner';
            this.doMission = () => {
                figureWhatToDo(topSecret); // we have access to topSecret
            };
        }
    }
```

In this example as well the data is 100% private and can't be reached outside the class, so our agent is safe.

Using naming conventions

We will decide that any property who is private will be prefixed with `_`.

Note that for this approach the data isn't really private.

```
export class SecretAgent{
    constructor(secret){
        this._topSecret = secret; // it private by convention
        this.coverStory = 'just a simple gardner';
        this.doMission = () => {
            figureWhatToDo(this_topSecret);
        };
    }
}
```

Section 22.9: Class Name binding

ClassDeclaration's Name is bound in different ways in different scopes -

1. The scope in which the class is defined - `let` binding
2. The scope of the class itself - within { and } in `class {}` - `const` binding

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

For example,

```
class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding
```

This is not the same for a Function -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```

Chapter 23: Namespacing

Section 23.1: Namespace by direct assignment

```
 //Before: antipattern 3 global variables
    var setActivePage = function () {};
    var getPage = function() {};
    var redirectPage = function() {};

//After:  just 1 global variable, no function collision and more meaningful function names
    var NavigationNs = NavigationNs || {};
    NavigationNs.active = function() {}
    NavigationNs.pagination = function() {}
    NavigationNs.redirection = function() {}
```

Section 23.2: Nested Namespaces

When multiple modules are involved, avoid proliferating global names by creating a single global namespace. From there, any sub-modules can be added to the global namespace. (Further nesting will slow down performance and add unnecessary complexity.) Longer names can be used if name clashes are an issue:

```
var NavigationNs = NavigationNs || {};
    NavigationNs.active =  {};
    NavigationNs.pagination = {};
    NavigationNs.redirection = {};

    // The second level start here.
    Navigational.pagination.jquery = function();
    Navigational.pagination.angular = function();
    Navigational.pagination.ember = function();
```

Chapter 24: Context (this)

Section 24.1: this with simple objects

```
var person = {
  name: 'John Doe',
  age: 42,
  gender: 'male',
  bio: function() {
    console.log('My name is ' + this.name);
  }
};
person.bio(); // logs "My name is John Doe"
var bio = person.bio;
bio(); // logs "My name is undefined"
```

In the above code, `person.bio` makes use of the **context** (`this`). When the function is called as `person.bio()`, the context gets passed automatically, and so it correctly logs "My name is John Doe". When assigning the function to a variable though, it loses its context.

In non-strict mode, the default context is the global object (`window`). In strict mode it is `undefined`.

Section 24.2: Saving this for use in nested functions / objects

One common pitfall is to try and use `this` in a nested function or an object, where the context has been lost.

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(function(result){
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Here the context (`this`) is lost in the inner callback function. To correct this, you can save the value of `this` in a variable:

```
document.getElementById('myAJAXButton').onclick = function(){
    var self = this;
    makeAJAXRequest(function(result){
      if (result) { // success
        self.className = 'success';
      }
    })
}
```

Version ≥ 6

ES6 introduced arrow functions which include lexical `this` binding. The above example could be written like this:

```
document.getElementById('myAJAXButton').onclick = function(){
    makeAJAXRequest(result => {
      if (result) { // success
        this.className = 'success';
      }
    })
}
```

Section 24.3: Binding function context

Version ≥ 5.1

Every function has a `bind` method, which will create a wrapped function that will call it with the correct context. See here for more information.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so value >
this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Hard binding

- The object of *hard binding* is to "hard" link a reference to **`this`**.
- Advantage: It's useful when you want to protect particular objects from being lost.
- Example:

```
function Person(){
    console.log("I'm " + this.name);
}

var person0 = {name: "Stackoverflow"}
var person1 = {name: "John"};
var person2 = {name: "Doe"};
var person3 = {name: "Ala Eddine JEBALI"};

var origin = Person;
Person = function(){
    origin.call(person0);
}

Person();
//outputs: I'm Stackoverflow

Person.call(person1);
//outputs: I'm Stackoverflow


Person.apply(person2);
```

```
//outputs: I'm Stackoverflow


Person.call(person3);
//outputs: I'm Stackoverflow
```

- So, as you can remark in the example above, whatever object you pass to *Person*, it'll always use *person0 object*: **it's hard binded**.

Section 24.4: this in constructor functions

When using a function as a constructor, it has a special `this` binding, which refers to the newly created object:

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}

var cat = new Cat("Tom"); // is a Cat object
cat.sound; // Returns "Meow"

var cat2 = Cat("Tom"); // is undefined -- function got executed in global context
window.name; // "Tom"
cat2.name; // error! cannot access property of undefined
```

Chapter 25: Setters and Getters

Setters and getters are object properties that call a function when they are set/gotten.

Section 25.1: Defining a Setter/Getter Using Object.defineProperty

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
    get: function(){
        return "a value";
    },
    set: function(value){
        setValue = value;
    }
});
```

Section 25.2: Defining an Setter/Getter in a Newly Created Object

JavaScript allows us to define getters and setters in the object literal syntax. Here's an example:

```
var date = {
    year: '2017',
    month: '02',
    day: '27',
    get date() {
        // Get the date in YYYY-MM-DD format
        return `${this.year}-${this.month}-${this.day}`
    },
    set date(dateString) {
        // Set the date from a YYYY-MM-DD formatted string
        var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;

        // Check that the string is correctly formatted
        if (dateRegExp.test(dateString)) {
            var parsedDate = dateRegExp.exec(dateString);
            this.year = parsedDate[1];
            this.month = parsedDate[2];
            this.day = parsedDate[3];
        }
        else {
            throw new Error('Date string must be in YYYY-MM-DD format');
        }
    }
};
```

Accessing the `date.date` property would return the value `2017-02-27`. Setting `date.date = '2018-01-02` would call the setter function, which would then parse the string and set `date.year = '2018'`, `date.month = '01'`, and `date.day = '02'`. Trying to pass an incorrectly formatted string (such as `"hello"`) would throw an error.

Section 25.3: Defining getters and setters in ES6 class

```
class Person {
  constructor(firstname, lastname) {
```

```
    this._firstname = firstname;
    this._lastname = lastname;
  }

  get firstname() {
    return this._firstname;
  }

  set firstname(name) {
    this._firstname = name;
  }

  get lastname() {
    return this._lastname;
  }

  set lastname(name) {
    this._lastname = name;
  }
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

Chapter 26: Events

Section 26.1: Page, DOM and Browser loading

This is an example to explain the variations of load events.

1. **onload event**

```
<body onload="someFunction()">
<img src="image1" />
<img src="image2" />
</body>

<script>
    function someFunction() {
    console.log("Hi! I am loaded");
}
</script>
```

In this case, the message is logged once *all the contents of the page including the images and stylesheets(if any)* are completely loaded.

2. **DOMContentLoaded event**

```
document.addEventListener("DOMContentLoaded", function(event) {
    console.log("Hello! I am loaded");
});
```

In the above code, the message is logged only after the DOM/document is loaded (*ie:once the DOM is constructed*).

3. **Self-invoking anonymous function**

```
(function(){
    console.log("Hi I am an anonymous function! I am loaded");
})();
```

Here, the message gets logged as soon as the browser interprets the anonymous function. It means, this function can get executed even before the DOM is loaded.

Chapter 27: Inheritance

Section 27.1: Standard function prototype

Start by defining a `Foo` function that we'll use as a constructor.

```
function Foo (){}
```

By editing `Foo.prototype`, we can define properties and methods that will be shared by all instances of `Foo`.

```
Foo.prototype.bar = function() {
  return 'I am bar';
};
```

We can then create an instance using the `new` keyword, and call the method.

```
var foo = new Foo();

console.log(foo.bar()); // logs `I am bar`
```

Section 27.2: Difference between Object.key and Object.prototype.key

Unlike in languages like Python, static properties of the constructor function are *not* inherited to instances. Instances only inherit from their prototype, which inherits from the parent type's prototype. Static properties are never inherited.

```
function Foo() {};
Foo.style = 'bold';

var foo = new Foo();

console.log(Foo.style); // 'bold'
console.log(foo.style); // undefined

Foo.prototype.style = 'italic';

console.log(Foo.style); // 'bold'
console.log(foo.style); // 'italic'
```

Section 27.3: Prototypal inheritance

Suppose we have a plain object called `prototype`:

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Now we want another object called `obj` that inherits from `prototype`, which is the same as saying that `prototype` is the prototype of `obj`

```
var obj = Object.create(prototype);
```

Now all the properties and methods from `prototype` will be available to `obj`

```
console.log(obj.foo);
```

```
console.log(obj.bar());
```

Console output

```
"foo"
"foo"
```

Prototypal inheritance is made through object references internally and objects are completely mutable. This means any change you make on a prototype will immediately affect every other object that prototype is prototype of.

```
prototype.foo = "bar";
console.log(obj.foo);
```

Console output

```
"bar"
```

`Object.prototype` is the prototype of every object, so it's strongly recommended you don't mess with it, especially if you use any third party library, but we can play with it a little bit.

```
Object.prototype.breakingLibraries = 'foo';
console.log(obj.breakingLibraries);
console.log(prototype.breakingLibraries);
```

Console output

```
"foo"
"foo"
```

Fun fact I've used the browser console to make these examples and broken this page by adding that `breakingLibraries` property.

Section 27.4: Pseudo-classical inheritance

It's an emulation of classical inheritance using prototypical inheritance which shows how powerful prototypes are. It was made to make the language more attractive to programmers coming from other languages.

Version < 6

IMPORTANT NOTE: Since ES6 it doesn't make sense to use pseudo-classical inheritance since the language simulates conventional classes. If you're not using ES6, you should. If you still want to use the classical inheritance pattern and you're in a ECMAScript 5 or lower environment, then pseudo-classical is your best bet.

A "class" is just a function that is made to be called with the **new** operand and it's used as a constructor.

```
function Foo(id, name) {
    this.id = id;
    this.name = name;
}

var foo = new Foo(1, 'foo');
```

```
console.log(foo.id);
```

Console output

> 1

foo is an instance of Foo. The JavaScript coding convention says if a function begins with a capital letter case it can be called as a constructor (with the `new` operand).

To add properties or methods to the "class" you have to add them to its prototype, which can be found in the `prototype` property of the constructor.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Console output

> bar

In fact what Foo is doing as a "constructor" is just creating objects with `Foo.prototype` as it's prototype.

You can find a reference to its constructor on every object

```
console.log(foo.constructor);
```

> function Foo(id, name) { ...

```
console.log({ }.constructor);
```

> function Object() { [native code] }

And also check if an object is an instance of a given class with the `instanceof` operator

```
console.log(foo instanceof Foo);
```

> true

```
console.log(foo instanceof Object);
```

> true

Section 27.5: Setting an Object's prototype

`Version ≥ 5`

With ES5+, the `Object.create` function can be used to create an Object with any other Object as it's prototype.

```
const anyObj = {
    hello() {
        console.log(`this.foo is ${this.foo}`);
    },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

To explicitly create an Object without a prototype, use `null` as the prototype. This means the Object will not inherit from `Object.prototype` either and is useful for Objects used for existence checking dictionaries, e.g.

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull ; // false
```

Version ≥ 6

From ES6, the prototype of an existing Object can be changed using `Object.setPrototypeOf`, for example

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

This can be done almost anywhere, including on a `this` object or in a constructor.

Note: This process is very slow in current browsers and should be used sparingly, try to create the Object with the desired prototype instead.

Version < 5

Before ES5, the only way to create an Object with a manually defined prototype was to construct it with `new`, for example

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

This behaviour is close enough to `Object.create` that it is possible to write a polyfill.

Chapter 28: Method Chaining

Section 28.1: Chainable object design and chaining

Chaining and Chainable is a design methodology used to design object behaviors so that calls to object functions return references to self, or another object, providing access to additional function calls allowing the calling statement to chain together many calls without the need to reference the variable holding the object/s.

Objects that can be chained are said to be chainable. If you call an object chainable, you should ensure that all returned objects / primitives are of the correct type. It only takes one time for your chainable object to not return the correct reference (easy to forget to add `return this`) and the person using your API will lose trust and avoid chaining. Chainable objects should be all or nothing (not a chainable object even if parts are). An object should not be called chainable if only some of its functions are.

Object designed to be chainable

```
function Vec(x = 0, y = 0){
    this.x = x;
    this.y = y;
    // the new keyword implicitly implies the return type
    // as this and thus is chainable by default.
}
Vec.prototype = {
    add : function(vec){
        this.x += vec.x;
        this.y += vec.y;
        return this; // return reference to self to allow chaining of function calls
    },
    scale : function(val){
        this.x *= val;
        this.y *= val;
        return this; //  return reference to self to allow chaining of function calls
    },
    log :function(val){
        console.log(this.x + ' : ' + this.y);
        return this;
    },
    clone : function(){
        return new Vec(this.x,this.y);
    }
}
```

Chaining example

```
var vec = new Vec();
vec.add({x:10,y:10})
    .add({x:10,y:10})
    .log()             // console output "20 : 20"
    .add({x:10,y:10})
    .scale(1/30)
    .log()             // console output "1 : 1"
    .clone()           // returns a new instance of the object
    .scale(2)          // from which you can continue chaining
    .log()
```

Don't create ambiguity in the return type

Not all function calls return a useful chainable type, nor do they always return a reference to self. This is where common sense use of naming is important. In the above example the function call `.clone()` is unambiguous. Other examples are `.toString()` implies a string is returned.

An example of an ambiguous function name in a chainable object.

```
// line object represents a line
line.rotate(1)
   .vec();  // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
   .asVec()    // unambiguous implies the return type is the line as a vec (vector)
   .add({x:10,y:10)
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Syntax convention

There is no formal usage syntax when chaining. The convention is to either chain the calls on a single line if short or to chain on the new line indented one tab from the referenced object with the dot on the new line. Use of the semicolon is optional but does help by clearly denoting the end of the chain.

```
vec.scale(2).add({x:2,y:2}).log();  // for short chains

vec.scale(2)     // or alternate syntax
    .add({x:2,y:2})
    .log();  // semicolon makes it clear the chain ends here

// and sometimes though not necessary
vec.scale(2)
    .add({x:2,y:2})
    .clone()    // clone adds a new reference to the chain
         .log(); // indenting to signify the new reference

// for chains in chains
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})  // a chain as an argument
         .add({x:2,y:2})      // is indented
         .scale(2))
    .log();

// or sometimes
vec.scale(2)
    .add({x:2,y:2})
    .add(vec1.add({x:2,y:2})  // a chain as an argument
         .add({x:2,y:2})      // is indented
         .scale(2)
    ).log();   // the argument list is closed on the new line
```

A bad syntax

```
 vec          // new line before the first function call
    .scale()  // can make it unclear what the intention is
    .log();

 vec.          // the dot on the end of the line
    scale(2).  // is very difficult to see in a mass of code
    scale(1/2); // and will likely frustrate as can easily be missed
               // when trying to locate bugs
```

Left hand side of assignment

When you assign the results of a chain the last returning call or object reference is assigned.

```
var vec2 = vec.scale(2)
                .add(x:1,y:10)
                .clone();   // the last returned result is assigned
                                // vec2 is a clone of vec after the scale and add
```

In the above example `vec2` is assigned the value returned from the last call in the chain. In this case, that would be a copy of `vec` after the scale and add.

Summary

The advantage of changing is clearer more maintainable code. Some people prefer it and will make chainable a requirement when selecting an API. There is also a performance benefit as it allows you to avoid having to create variables to hold interim results. With the last word being that chainable objects can be used in a conventional way as well so you don't enforce chaining by making an object chainable.

Section 28.2: Method Chaining

Method chaining is a programming strategy that simplifies your code and beautifies it. Method chaining is done by ensuring that each method on an object returns the entire object, instead of returning a single element of that object. For example:

```
function Door() {
    this.height = '';
    this.width = '';
    this.status = 'closed';
}

Door.prototype.open = function() {
    this.status = 'opened';
    return this;
}

Door.prototype.close = function() {
    this.status = 'closed';
    return this;
}
Door.prototype.setParams = function(width,height) {
    this.width = width;
    this.height = height;
    return this;
}

Door.prototype.doorStatus = function() {
    console.log('The',this.width,'x',this.height,'Door is',this.status);
    return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Note that each method in `Door.prototype` returns `this`, which refers to the entire instance of that `Door` object.

Chapter 29: Callbacks

Section 29.1: Simple Callback Usage Examples

Callbacks offer a way to extend the functionality of a function (or method) ***without changing*** its code. This approach is often used in modules (libraries / plugins), the code of which is not supposed to be changed.

Suppose we have written the following function, calculating the sum of a given array of values:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

Now suppose that we want to do something with each value of the array, e.g. display it using `alert()`. We could make the appropriate changes in the code of `foo`, like this:

```
function foo(array) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        alert(array[i]);
        sum += array[i];
    }
    return sum;
}
```

But what if we decide to use `console.log` instead of `alert()`? Obviously changing the code of `foo`, whenever we decide to do something else with each value, is not a good idea. It is much better to have the option to change our mind without changing the code of `foo`. That's exactly the use case for callbacks. We only have to slightly change `foo`'s signature and body:

```
function foo(array, callback) {
    var sum = 0;
    for (var i = 0; i < array.length; i++) {
        callback(array[i]);
        sum += array[i];
    }
    return sum;
}
```

And now we are able to change the behaviour of `foo` just by changing its parameters:

```
var array = [];
foo(array, alert);
foo(array, function (x) {
    console.log(x);
});
```

Examples with Asynchronous Functions

In jQuery, the `$.getJSON()` method to fetch JSON data is asynchronous. Therefore, passing code in a callback makes sure that the code is called *after* the JSON is fetched.

`$.getJSON()` syntax:

```
$.getJSON( url, dataObject, successCallback );
```

Example of `$.getJSON()` code:

```
$.getJSON("foo.json", {}, function(data) {
    // data handling code
});
```

The following would *not* work, because the data-handling code would likely be called *before* the data is actually received, because the `$.getJSON` function takes an unspecified length of time and does not hold up the call stack as it waits for the JSON.

```
$.getJSON("foo.json", {});
// data handling code
```

Another example of an asynchronous function is jQuery's `animate()` function. Because it takes a specified time to run the animation, sometimes it is desirable to run some code directly following the animation.

`.animate()` syntax:

```
jQueryElement.animate( properties, duration, callback );
```

For example, to create a fading-out animation after which the element completely disappears, the following code can be run. Note the use of the callback.

```
elem.animate( { opacity: 0 }, 5000, function() {
    elem.hide();
} );
```

This allows the element to be hidden right after the function has finished execution. This differs from:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

because the latter does not wait for `animate()` (an asynchronous function) to complete, and therefore the element is hidden right away, producing an undesirable effect.

Section 29.2: Continuation (synchronous and asynchronous)

Callbacks can be used to provide code to be executed after a method has completed:

```
/**
 * @arg {Function} then continuation callback
 */
function doSomething(then) {
  console.log('Doing something');
  then();
}

// Do something, then execute callback to log 'done'
doSomething(function () {
  console.log('Done');
});
```

```
console.log('Doing something else');

// Outputs:
//   "Doing something"
//   "Done"
//   "Doing something else"
```

The `doSomething()` method above executes synchronously with the callback - execution blocks until `doSomething()` returns, ensuring that the callback is executed before the interpreter moves on.

Callbacks can also be used to execute code asynchronously:

```
doSomethingAsync(then) {
  setTimeout(then, 1000);
  console.log('Doing something asynchronously');
}

doSomethingAsync(function() {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something asynchronously"
//   "Doing something else"
//   "Done"
```

The `then` callbacks are considered continuations of the `doSomething()` methods. Providing a callback as the last instruction in a function is called a [tail-call](), which is [optimized by ES2015 interpreters]().

Section 29.3: What is a callback?

This is a normal function call:

```
console.log("Hello World!");
```

When you call a normal function, it does its job and then returns control back to the caller.

However, sometimes a function needs to return control back to the caller in order to do its job:

```
[1,2,3].map(function double(x) {
    return 2 * x;
});
```

In the above example, the function `double` is a callback for the function `map` because:

1. The function `double` is given to the function `map` by the caller.
2. The function `map` needs to call the function `double` zero or more times in order to do its job.

Thus, the function `map` is essentially returning control back to the caller every time it calls the function `double`. Hence, the name “callback”.

Functions may accept more than one callback:

```
promise.then(function onFulfilled(value) {
    console.log("Fulfilled with value " + value);
```

```
}, function onRejected(reason) {
    console.log("Rejected with reason " + reason);
});
```

Here then function `then` accepts two callback functions, `onFulfilled` and `onRejected`. Furthermore, only one of these two callback functions is actually called.

What's more interesting is that the function `then` returns before either of the callbacks are called. Hence, a callback function may be called even after the original function has returned.

Section 29.4: Callbacks and `this`

Often when using a callback you want access to a specific context.

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg);  // <= will fail because "this" is undefined
  });
}

var s = new SomeClass("hello", someElement);
```

Solutions

- Use `bind`

  `bind` effectively generates a new function that sets **`this`** to whatever was passed to `bind` then calls the original function.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click', function() {
      console.log(this.msg);
    }.bind(this));  // <=-  bind the function to `this`
  }
  ```

- Use arrow functions

  Arrow functions automatically bind the current **`this`** context.

  ```
  function SomeClass(msg, elem) {
    this.msg = msg;
    elem.addEventListener('click',() => {   // <=-  arrow function binds `this`
      console.log(this.msg);
    });
  }
  ```

Often you'd like to call a member function, ideally passing any arguments that were passed to the event on to the function.

Solutions:

- Use bind

  ```
  function SomeClass(msg, elem) {
  ```

```
    this.msg = msg;
    elem.addEventListener('click', this.handleClick.bind(this));
  }

  SomeClass.prototype.handleClick = function(event) {
    console.log(event.type, this.msg);
  };
```

- Use arrow functions and the rest operator

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', (...a) => this.handleClick(...a));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- For DOM event listeners in particular you can implement the [EventListener interface]

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
     fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};
```

Section 29.5: Callback using Arrow function

Using arrow function as callback function can reduce lines of code.

The default syntax for arrow function is

```
() => {}
```

This can be used as callbacks

For example if we want to print all elements in an array [1,2,3,4,5]

without arrow function, the code will look like this

```
[1,2,3,4,5].forEach(function(x){
                 console.log(x);
            }
```

With arrow function, it can be reduced to

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Here the callback function `function(x){console.log(x)}` is reduced to `x=>console.log(x)`

Section 29.6: Error handling and control-flow branching

Callbacks are often used to provide error handling. This is a form of control flow branching, where some instructions are executed only when an error occurs:

```
const expected = true;

function compare(actual, success, failure) {
  if (actual === expected) {
    success();
  } else {
    failure();
  }
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

Code execution in `compare()` above has two possible branches: `success` when the expected and actual values are the same, and `error` when they are different. This is especially useful when control flow should branch after some asynchronous instruction:

```
function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
//   "Doing something else"
//   "Value was expected"
//   "Value was unexpected/exceptional"
```

It should be noted, multiple callbacks do not have to be mutually exclusive – both methods could be called. Similarly, the `compare()` could be written with callbacks that are optional (by using a noop as the default value - see Null Object pattern).

Chapter 30: Intervals and Timeouts

Section 30.1: Recursive setTimeout

To repeat a function indefinitely, `setTimeout` can be called recursively:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}

setTimeout(repeatingFunc, 5000);
```

Unlike `setInterval`, this ensures that the function will execute even if the function's running time is longer than the specified delay. However, it does not guarantee a regular interval between function executions. This behaviour also varies because an exception before the recursive call to `setTimeout` will prevent it from repeating again, while `setInterval` would repeat indefinitely regardless of exceptions.

Section 30.2: Intervals

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

window.setInterval(waitFunc,5000);
```

Section 30.3: Intervals

Standard

You don't need to create the variable, but it's a good practice as you can use that variable with clearInterval to stop the currently running interval.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

If you need to pass parameters to the doSomething function, you can pass them as additional parameters beyond the first two to setInterval.

Without overlapping

setInterval, as above, will run every 5 seconds (or whatever you set it to) no matter what. Even if the function doSomething takes long than 5 seconds to run. That can create issues. If you just want to make sure there is that pause in between runnings of doSomething, you can do this:

```
(function(){

   doSomething();

   setTimeout(arguments.callee, 5000);

})()
```

Section 30.4: Removing intervals

`window.setInterval()` returns an `IntervalID`, which can be used to stop that interval from continuing to run. To do this, store the return value of `window.setInterval()` in a variable and call `clearInterval()` with that variable as the only argument:

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc,5000);

window.setTimeout(function(){
    clearInterval(interval);
},32000);
```

This will log `This will be logged every 5 seconds` every 5 seconds, but will stop it after 32 seconds. So it will log the message 6 times.

Section 30.5: Removing timeouts

`window.setTimout()` returns a `TimeoutID`, which can be used to stop that timeout from running. To do this, store the return value of `window.setTimeout()` in a variable and call `clearTimeout()` with that variable as the only argument:

```
function waitFunc(){
    console.log("This will not be logged after 5 seconds");
}
function stopFunc(){
    clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc,5000);
window.setTimeout(stopFunc,3000);
```

This will not log the message because the timer is stopped after 3 seconds.

Section 30.6: setTimeout, order of operations, clearTimeout

setTimeout

- Executes a function, after waiting a specified number of milliseconds.
- used to delay the execution of a function.

Syntax : `setTimeout(function, milliseconds)` or `window.setTimeout(function, milliseconds)`

Example : This example outputs "hello" to the console after 1 second. The second parameter is in milliseconds, so 1000 = 1 sec, 250 = 0.25 sec, etc.

```
setTimeout(function() {
    console.log('hello');
}, 1000);
```

Problems with setTimeout

if you're using the `setTimeout` method in a for loop **:**

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

This will output the value `3` `three` times, which is not correct.

Workaround of this problem :

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

It will output the value `0`,1,2. Here, we're passing the `i` into the function as a parameter(`j`).

Order of operations

Additionally though, due to the fact that JavaScript is single threaded and uses a global event loop, `setTimeout` can be used to add an item to the end of the execution queue by calling `setTimeout` with zero delay. For example:

```
setTimeout(function() {
    console.log('world');
}, 0);

console.log('hello');
```

Will actually output:

```
hello
world
```

Also, zero milliseconds here does not mean the function inside the setTimeout will execute immediately. It will take slightly more than that depending upon the items to be executed remaining in the execution queue. This one is just pushed to the end of the queue.

Cancelling a timeout

clearTimeout() : stops the execution of the function specified in `setTimeout()`

Syntax : clearTimeout(timeoutVariable) or window.clearTimeout(timeoutVariable)

Example :

```
var timeout = setTimeout(function() {
    console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

Chapter 31: Regular expressions

Flags	Details
g	**g**lobal. All matches (don't return on the first match).
m	**m**ulti-line. Causes **^** & **$** to match the begin/end of each line (not only begin/end of string).
i	**i**nsensitive. Case insensitive match (ignores case of [a-zA-Z]).
u	**u**nicode : Pattern strings are treated as **UTF-16**. Also causes escape sequences to match Unicode characters.
y	stick**y**: matches only from the index indicated by the lastIndex property of this regular expression in the target string (and does not attempt to match from any later indexes).

Section 31.1: Creating a RegExp Object

Standard Creation

It is recommended to use this form only when creating regex from dynamic variables.

Use when the expression may change or the expression is user generated.

```
var re = new RegExp(".*");
```

With flags:

```
var re = new RegExp(".*", "gmi");
```

With a backslash: (this must be escaped because the regex is specified with a string)

```
var re = new RegExp("\\w*");
```

Static initialization

Use when you know the regular expression will not change, and you know what the expression is before runtime.

```
var re = /.*/;
```

With flags:

```
var re = /.*/gmi;
```

With a backslash: (this should not be escaped because the regex is specified in a literal)

```
var re = /\w*/;
```

Section 31.2: RegExp Flags

There are several flags you can specify to alter the RegEx behaviour. Flags may be appended to the end of a regex literal, such as specifying `gi` in `/test/gi`, or they may be specified as the second argument to the `RegExp` constructor, as in `new RegExp('test', 'gi')`.

`g` - Global. Finds all matches instead of stopping after the first.

`i` - Ignore case. `/[a-z]/i` is equivalent to `/[a-zA-Z]/`.

`m` - Multiline. `^` and `$` match the beginning and end of each line respectively treating `\n` and `\r` as delimiters instead of simply the beginning and end of the entire string.

Version ≥ 6

`u` - Unicode. If this flag is not supported you must match specific Unicode characters with `\uXXXX` where `XXXX` is the character's value in hexadecimal.

`y` - Finds all consecutive/adjacent matches.

Section 31.3: Check if string contains pattern using .test()

```
var re = /[a-z]+/;
if (re.test("foo")) {
    console.log("Match exists.");
}
```

The `test` method performs a search to see if a regular expression matches a string. The regular expression `[a-z]+` will search for one or more lowercase letters. Since the pattern matches the string, "match exists" will be logged to the console.

Section 31.4: Matching With .exec()

Match Using `.exec()`

`RegExp.prototype.exec(string)` returns an array of captures, or `null` if there was no match.

```
var re = /([0-9]+)[a-z]+/;
var match = re.exec("foo123bar");
```

`match.index` is 3, the (zero-based) location of the match.

`match[0]` is the full match string.

`match[1]` is the text corresponding to the first captured group. `match[n]` would be the value of the *n*th captured group.

Loop Through Matches Using `.exec()`

```
var re = /a/g;
var result;
while ((result = re.exec('barbatbaz')) !== null) {
    console.log("found '" + result[0] + "', next exec starts at index '" + re.lastIndex + "'");
}
```

Expected output

> found 'a', next exec starts at index '2'
> found 'a', next exec starts at index '5'
> found 'a', next exec starts at index '8'

Section 31.5: Using RegExp With Strings

The String object has the following methods that accept regular expressions as arguments.

- `"string".match(...`
- `"string".replace(...`
- `"string".split(...`
- `"string".search(...`

Match with RegExp

```
console.log("string".match(/[i-n]+/));
console.log("string".match(/(r)[i-n]+/));
```

Expected output

> Array ["in"]
> Array ["rin", "r"]

Replace with RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Expected output

> strfoog

Split with RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Expected output

> Array ["str", "gstr", "g"]

Search with RegExp

`.search()` returns the index at which a match is found or -1.

```
console.log("string".search(/[i-n]+/));
console.log("string".search(/[o-q]+/));
```

Expected output

> 3
> -1

Section 31.6: RegExp Groups

JavaScript supports several types of group in its Regular Expressions, *capture groups*, *non-capture groups* and *look-aheads*. Currently, there is no *look-behind* support.

Capture

Sometimes the desired match relies on its context. This means a simple *RegExp* will over-find the piece of the *String* that is of interest, so the solution is to write a capture group `(pattern)`. The captured data can then be referenced as...

- String replacement `"$n"` where `n` is the *n th* capture group (starting from `1`)
- The *n th* argument in a callback function
- If the *RegExp* is not flagged `g`, the *n+1 th* item in a returned `str.match` *Array*
- If the *RegExp* is flagged `g`, `str.match` discards captures, use `re.exec` instead

Say there is a *String* where all `+` signs need to be replaced with a space, but only if they follow a letter character. This means a simple match would include that letter character and it would also be removed. Capturing it is the solution as it means the matched letter can be preserved.

```
let str = "aa+b+cc+1+2",
    re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Non-Capture

Using the form `(?:pattern)`, these work in a similar way to capture groups, except they do not store the contents of the group after the match.

They can be particularly useful if other data is being captured which you don't want to move the indices of, but need to do some advanced pattern matching such as an OR

```
let str = "aa+b+cc+1+2",
    re = /(?:\b|c)([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Look-Ahead

If the desired match relies on something which follows it, rather than matching that and capturing it, it is possible to use a look-ahead to test for it but not include it in the match. A positive look-ahead has the form `(?=pattern)`, a negative look-ahead (where the expression match only happens if the look-ahead pattern did not match) has the form `(?!pattern)`

```
let str = "aa+b+cc+1+2",
    re = /\+(?=[a-z])/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Section 31.7: Replacing string match with a callback function

`String#replace` can have a function as its second argument so you can provide a replacement based on some logic.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
```

```
  }
});
// will return Start string End
```

One line template library

```
let data = {name: 'John', surname: 'Doe'}
"My name is {surname}, {name} {surname}".replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);

// "My name is Doe, John Doe"
```

Section 31.8: Using Regex.exec() with parentheses regex to extract matches of a string

Sometimes you doesn't want to simply replace or remove the string. Sometimes you want to extract and process matches. Here an example of how you manipulate matches.

What is a match ? When a compatible substring is found for the entire regex in the string, the exec command produce a match. A match is an array compose by firstly the whole substring that matched and all the parenthesis in the match.

Imagine a html string :

```
<html>
<head></head>
<body>
  <h1>Example</h1>
  <p>Look at this great link : <a href="http://goalkicker.com">goalkicker</a>
http://anotherlinkoutsideatag</p>
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
</body>
```

You want to extract and get all the links inside an a tag. At first, here the regex you write :

```
var re = /<a[^>]*href="https?:\/\/.*"[^>]*>[^<]*<\/a>/g;
```

But now, imagine you want the href and the anchor of each link. And you want it together. You can simply add a new regex in for each match **OR** you can use parentheses :

```
var re = /<a[^>]*href="(https?:\/\/.*)"[^>]*>([^<]*)<\/a>/g;
var str = '<html>\n    <head></head>\n    <body>\n        <h1>Example</h1>\n        <p>Look at this
great link: <a href="http://goalkicker.com">goalkicker</a> http://anotherlinkoutsideatag</p>\n\n
     Copyright <a href="https://stackoverflow.com">Stackoverflow</a>\n    </body>\';\n';
var m;
var links = [];

while ((m = re.exec(str)) !== null) {
    if (m.index === re.lastIndex) {
        re.lastIndex++;
    }
    console.log(m[0]); // The all substring
    console.log(m[1]); // The href subpart
    console.log(m[2]); // The anchor subpart

    links.push({
      match : m[0],   // the entire match
      href : m[1],    // the first parenthesis => (https?:\/\/.*)
```

```
        anchor : m[2],  // the second one => ([^<]*)
    });
}
```

At the end of the loop, you have an array of link with `anchor` and `href` and you can use it to write markdown for example :

```
links.forEach(function(link) {
  console.log('[%s](%s)', link.anchor, link.href);
});
```

To go further :

- Nested parenthesis

Chapter 32: Cookies

Section 32.1: Test if cookies are enabled

If you want to make sure cookies are enabled before using them, you can use `navigator.cookieEnabled`:

```
if (navigator.cookieEnabled === false)
{
    alert("Error: cookies not enabled!");
}
```

Note that on older browsers `navigator.cookieEnabled` may not exist and be undefined. In those cases you won't detect that cookies are not enabled.

Section 32.2: Adding and Setting Cookies

The following variables set up the below example:

```
var COOKIE_NAME = "Example Cookie";    /* The cookie's name. */
var COOKIE_VALUE = "Hello, world!";    /* The cookie's value. */
var COOKIE_PATH = "/foo/bar";          /* The cookie's path. */
var COOKIE_EXPIRES;                    /* The cookie's expiration date (config'd below). */

/* Set the cookie expiration to 1 minute in future (60000ms = 1 minute). */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();
```

```
document.cookie +=
  COOKIE_NAME + "=" + COOKIE_VALUE
  + "; expires=" + COOKIE_EXPIRES
  + "; path=" + COOKIE_PATH;
```

Section 32.3: Reading cookies

```
var name = name + "=",
    cookie_array = document.cookie.split(';'),
    cookie_value;
for(var i=0;i<cookie_array.length;i++) {
    var cookie=cookie_array[i];
    while(cookie.charAt(0)==' ')
        cookie = cookie.substring(1,cookie.length);
    if(cookie.indexOf(name)==0)
        cookie_value = cookie.substring(name.length,cookie.length);
    }
```

This will set `cookie_value` to the value of the cookie, if it exists. If the cookie is not set, it will set `cookie_value` to **`null`**

Section 32.4: Removing cookies

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "=; expires=" + expiry.toGMTString() + "; path=/"
```

This will remove the cookie with a given `name`.

Chapter 33: Web Storage

Parameter	Description
name	The key/name of the item
value	The value of the item

Section 33.1: Using localStorage

The localStorage object provides persistent (but not permanent - see limits below) key-value storage of strings. Any changes are immediately visible in all other windows/frames from the same origin. The stored values persistent indefinitely unless the user clears saved data or configures an expiration limit. localStorage uses a map-like interface for getting and setting values.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')); // "John Smith"

localStorage.removeItem('name');
console.log(localStorage.getItem('name')); // null
```

If you want to store simple structured data, you can use JSON to serialize it to and from strings for storage.

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));

console.log(JSON.parse(localStorage.getItem('players')));
// [ Object { name: "Tyler", score: 22 }, Object { name: "Ryan", score: 41 } ]
```

localStorage limits in browsers

Mobile browsers:

Browser	Google Chrome	Android Browser	Firefox	iOS Safari
Version	40	4.3	34	6-8
Space available	10MB	2MB	10MB	5MB

Desktop browsers:

Browser	Google Chrome	Opera	Firefox	Safari	Internet Explorer
Version	40	27	34	6-8	9-11
Space available	10MB	10MB	10MB	5MB	10MB

Section 33.2: Simpler way of handling Storage

`localStorage`, `sessionStorage` are JavaScript **Objects** and you can treat them as such.
Instead of using Storage Methods like `.getItem()`, `.setItem()`, etc… here's a simpler alternative:

```
// Set
localStorage.greet = "Hi!"; // Same as: window.localStorage.setItem("greet", "Hi!");

// Get
localStorage.greet;         // Same as: window.localStorage.getItem("greet");

// Remove item
delete localStorage.greet;  // Same as: window.localStorage.removeItem("greet");
```

```
// Clear storage
localStorage.clear();
```

Example:

```
// Store values (Strings, Numbers)
localStorage.hello = "Hello";
localStorage.year  = 2017;

// Store complex data (Objects, Arrays)
var user = {name:"John", surname:"Doe", books:["A","B"]};
localStorage.user = JSON.stringify( user );

// Important: Numbers are stored as String
console.log( typeof localStorage.year ); // String

// Retrieve values
var someYear = localStorage.year; // "2017"

// Retrieve complex data
var userData = JSON.parse( localStorage.user );
var userName = userData.name; // "John"

// Remove specific data
delete localStorage.year;

// Clear (delete) all stored data
localStorage.clear();
```

Section 33.3: Storage events

Whenever a value in set in localStorage, a `storage` event will be dispatched on all other `windows` from the same origin. This can be used to synchronize state between different pages without reloading or communicating with a server. For example, we can reflect the value of an input element as paragraph text in another window:

First Window

```
var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Second Window

```
var output = document.createElement('p');
document.body.appendChild(output);

output.textContent = localStorage.getItem('user-value');

window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

Notes

Event is not fired or catchable under Chrome, Edge and Safari if domain was modified through script.

First window

```
// page url: http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Second Window

```
// page url: http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// Listener will never called under Chrome(53), Edge and Safari(10.0).
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

Section 33.4: sessionStorage

The sessionStorage object implements the same Storage interface as localStorage. However, instead of being shared with all pages from the same origin, sessionStorage data is stored separately for every window/tab. Stored data persists between pages *in that window/tab* for as long as it's open, but is visible nowhere else.

```
var audio = document.querySelector('audio');

// Maintain the volume if the user clicks a link then navigates back here.
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
  sessionStorage.setItem('volume', audio.volume);
};
```

Save data to sessionStorage

```
sessionStorage.setItem('key', 'value');
```

Get saved data from sessionStorage

```
var data = sessionStorage.getItem('key');
```

Remove saved data from sessionStorage

```
sessionStorage.removeItem('key')
```

Section 33.5: localStorage length

`localStorage.length` property returns an integer number indicating the number of elements in the `localStorage`

Example:

Set Items

```
localStorage.setItem('StackOverflow', 'Documentation');
localStorage.setItem('font', 'Helvetica');
localStorage.setItem('image', 'sprite.svg');
```

Get length

```
localStorage.length; // 3
```

Section 33.6: Error conditions

Most browsers, when configured to block cookies, will also block `localStorage`. Attempts to use it will result in an exception. Do not forget to manage these cases.

```
var video = document.querySelector('video')
try {
    video.volume = localStorage.getItem('volume')
} catch (error) {
    alert('If you\'d like your volume saved, turn on cookies')
}
video.play()
```

If error were not handled, program would stop functioning properly.

Section 33.7: Clearing storage

To clear the storage, simply run

```
localStorage.clear();
```

Section 33.8: Remove Storage Item

To remove a specific item from the browser Storage (the opposite of `setItem`) use `removeItem`

```
localStorage.removeItem("greet");
```

Example:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null
```

(Same applies for `sessionStorage`)

Chapter 34: Data attributes

Section 34.1: Accessing data attributes

Using the dataset property

The new `dataset` property allows access (for both reading and writing) to all data attributes `data-*` on any element.

```
<p>Countries:</p>
<ul>
  <li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">USA</li>
  <li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">Canada</li>
  <li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">France</li>
</ul>
<button type="button" onclick="correctDetails()">Correct Country Details</button>
<script>
function showDetails(item) {
    var msg = item.innerHTML
            + "\r\nISO ID: " + item.dataset.id
            + "\r\nDial Code: " + item.dataset.dialCode;
    alert(msg);
}

function correctDetails(item) {
    var item = document.getEmementById("C3");
    item.dataset.id = "FR";
    item.dataset.dialCode = "33";
}
</script>
```

Note: The `dataset` property is only supported in modern browsers and it's slightly slower than the `getAttribute` and `setAttribute` methods which are supported by all browsers.

Using the getAttribute & setAttribute methods

If you want to support the older browsers before HTML5, you can use the `getAttribute` and `setAttribute` methods which are used to access any attribute including the data attributes. The two functions in the example above can be written this way:

```
<script>
function showDetails(item) {
    var msg = item.innerHTML
            + "\r\nISO ID: " + item.getAttribute("data-id")
            + "\r\nDial Code: " + item.getAttribute("data-dial-code");
    alert(msg);
}

function correctDetails(item) {
    var item = document.getEmementById("C3");
    item.setAttribute("id", "FR");
    item.setAttribute("data-dial-code", "33");
}
</script>
```

Chapter 35: JSON

Parameter	Details
JSON.parse	**Parse a JSON string**
`input(string)`	JSON string to be parsed.
`reviver(function)`	Prescribes a transformation for the input JSON string.
JSON.stringify	**Serialize a serializable value**
`value(string)`	Value to be serialized according to the JSON specification.
`replacer(function` or `String[]` or `Number[])`	Selectively includes certain properties of the `value` object.
`space(String` or `Number)`	If a `number` is provided, then `space` number of whitespaces will be inserted of readability. If a `string` is provided, the string (first 10 characters) will be used as whitespaces.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write and easy for machines to parse and generate. It is important to realize that, in JavaScript, JSON is a string and not an object.

A basic overview can be found on the json.org website which also contains links to implementations of the standard in many different programming languages.

Section 35.1: JSON versus JavaScript literals

JSON stands for "JavaScript Object Notation", but it's not JavaScript. Think of it as just a *data serialization format* that *happens* to be directly usable as a JavaScript literal. However, it is not advisable to directly run (i.e. through `eval()`) JSON that is fetched from an external source. Functionally, JSON isn't very different from XML or YAML – some confusion can be avoided if JSON is just imagined as some serialization format that looks very much like JavaScript.

Even though the name implies just objects, and even though the majority of use cases through some kind of API always happen to be objects and arrays, JSON is not for just objects or arrays. The following primitive types are supported:

- String (e.g. `"Hello World!"`)
- Number (e.g. `42`)
- Boolean (e.g. `true`)
- The value `null`

`undefined` is not supported in the sense that an undefined property will be omitted from JSON upon serialization. Therefore, there is no way to deserialize JSON and end up with a property whose value is `undefined`.

The string `"42"` is valid JSON. JSON doesn't always have to have an outer envelope of `"{...}"` or `"[...]"`.

While some JSON is also valid JavaScript and some JavaScript is also valid JSON, there are some subtle differences between both languages and neither language is a subset of the other.

Take the following JSON string as an example:

```
{"color": "blue"}
```

This can be directly inserted into JavaScript. It will be syntactically valid and will yield the correct value:

```
const skin = {"color": "blue"};
```

However, we know that "color" is a valid identifier name and the quotes around the property name can be omitted:

```
const skin = {color: "blue"};
```

We also know that we can use single quotes instead of double quotes:

```
const skin = {'color': 'blue'};
```

But, if we were to take both of these literals and treat them as JSON, **neither will be syntactically valid** JSON:

```
{color: "blue"}
{'color': 'blue'}
```

JSON strictly requires all property names to be double quoted and string values to be double quoted as well.

It's common for JSON-newcomers to attempt to use code excerpts with JavaScript literals as JSON, and scratch their heads about the syntax errors they are getting from the JSON parser.

More confusion starts arising when *incorrect terminology* is applied in code or in conversation.

A common anti-pattern is to name variables that hold non-JSON values as "json":

```
fetch(url).then(function (response) {
  const json = JSON.parse(response.data); // Confusion ensues!

  // We're done with the notion of "JSON" at this point,
  // but the concept stuck with the variable name.
});
```

In the above example, `response.data` is a JSON string that is returned by some API. JSON stops at the HTTP response domain. The variable with the "json" misnomer holds just a JavaScript value (could be an object, an array, or even a simple number!)

A less confusing way to write the above is:

```
fetch(url).then(function (response) {
  const value = JSON.parse(response.data);

  // We're done with the notion of "JSON" at this point.
  // You don't talk about JSON after parsing JSON.
});
```

Developers also tend to throw the phrase "JSON object" around a lot. This also leads to confusion. Because as mentioned above, a JSON string doesn't have to hold an object as a value. "JSON string" is a better term. Just like "XML string" or "YAML string". You get a string, you parse it, and you end up with a value.

Section 35.2: Parsing with a reviver function

A reviver function can be used to filter or transform the value being parsed.

Version ≥ 5.1

```
var jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

var data = JSON.parse(jsonString, function reviver(key, value) {
  return key === 'name' ? value.toUpperCase() : value;
});
```

Version ≥ 6

```
const jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

This produces the following result:

```
[
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

This is particularly useful when data must be sent that needs to be serialized/encoded when being transmitted with JSON, but one wants to access it deserialized/decoded. In the following example, a date was encoded to its ISO 8601 representation. We use the reviver function to parse this in a JavaScript `Date`.

Version ≥ 5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
});
```

Version ≥ 6

```
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'date' ? new Date(value) : value
);
```

It is important to make sure the reviver function returns a useful value at the end of each iteration. If the reviver function returns **`undefined`**, no value or the execution falls off towards the end of the function, the property is deleted from the object. Otherwise, the property is redefined to be the return value.

Section 35.3: Serializing a value

A JavaScript value can be converted to a JSON string using the `JSON.stringify` function.

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` The value to convert to a JSON string.

```
/* Boolean */  JSON.stringify(true)             // 'true'
/* Number  */  JSON.stringify(12)               // '12'
/* String  */  JSON.stringify('foo')            // '"foo"'
/* Object  */  JSON.stringify({})               // '{}'
               JSON.stringify({foo: 'baz'})     // '{"foo": "baz"}'
/* Array   */  JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* Date    */  JSON.stringify(new Date())       // '"2016-08-06T17:25:23.588Z"'
/* Symbol  */  JSON.stringify({x:Symbol()})     // '{}'
```

2. `replacer` A function that alters the behaviour of the stringification process or an array of String and Number objects that serve as a whitelist for filtering the properties of the value object to be included in the JSON string. If this value is null or is not provided, all properties of the object are included in the resulting JSON string.

```
// replacer as a function
function replacer (key, value) {
    // Filtering out properties
    if (typeof value === "string") {
        return
    }
    return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'
```

```
// replacer as an array
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// only the `foundation`, `week`, and `month` properties are kept
```

3. `space` For readability, the number of spaces used for indentation may be specified as the third parameter.

```
JSON.stringify({x: 1, y: 1}, null, 2)  // 2 space characters will be used for indentation
/* output:
    {
      'x': 1,
      'y': 1
    }
*/
```

Alternatively, a string value can be provided to use for indentation. For example, passing `'\t'` will cause the tab character to be used for indentation.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
    {
        'x': 1,
        'y': 1
    }
*/
```

Section 35.4: Serializing and restoring class instances

You can use a custom `toJSON` method and reviver function to transmit instances of your own class in JSON. If an object has a `toJSON` method, its result will be serialized instead of the object itself.

Version < 6

```
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
```

```
    speed: this.speed
  };
};

Car.fromJSON = function(data) {
  return new Car(data.color, data.speed);
};
```

Version ≥ 6

```
class Car {
  constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
  }

  toJSON() {
    return {
      $type: 'com.example.Car',
      color: this.color,
      speed: this.speed
    };
  }

  static fromJSON(data) {
    return new Car(data.color, data.speed);
  }
}
```

```
var userJson = JSON.stringify({
  name: "John",
  car: new Car('red', 'fast')
});
```

This produces the a string with the following content:

```
{"name":"John","car":{"$type":"com.example.Car","color":"red","speed":"fast"}}
```

```
var userObject = JSON.parse(userJson, function reviver(key, value) {
  return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});
```

This produces the following object:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

Section 35.5: Serializing with a replacer function

A `replacer` function can be used to filter or transform values being serialized.

```
const userRecords = [
  {name: "Joe", points: 14.9, level: 31.5},
  {name: "Jane", points: 35.5, level: 74.4},
  {name: "Jacob", points: 18.5, level: 41.2},
```

```
  {name: "Jessie", points: 15.1, level: 28.1},
];

// Remove names and round numbers to integers to anonymize records before sharing
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);
```

This produces the following string:

```
'[{"points":14,"level":31},{"points":35,"level":74},{"points":18,"level":41},{"points":15,"level":2
8}]'
```

Section 35.6: Parsing a simple JSON string

The `JSON.parse()` method parses a string as JSON and returns a JavaScript primitive, array or object:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Section 35.7: Cyclic object values

Not all objects can be converted to a JSON string. When an object has cyclic self-references, the conversion will fail.

This is typically the case for hierarchical data structures where parent and child both reference each other:

```
const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name":"World","regions":[{"name":"North America","parent":"America"}]}

world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON
```

As soon as the process detects a cycle, the exception is raised. If there were no cycle detection, the string would be infinitely long.

Chapter 36: AJAX

AJAX stands for "Asynchronous JavaScript and XML". Although the name includes XML, JSON is more often used due to its simpler formatting and lower redundancy. AJAX allows the user to communicate with external resources without reloading the webpage.

Section 36.1: Sending and Receiving JSON Data via POST

Version ≥ 6

Fetch request promises initially return Response objects. These will provide response header information, but they don't directly include the response body, which may not have even loaded yet. Methods on the Response object such as `.json()` can be used to wait for the response body to load, then parse it.

```
const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("Got non-2XX response from API server.");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("Known users: ", users);
}, error => {
  console.error("Failed to fetch users due to error: ", error);
});
```

Section 36.2: Add an AJAX preloader

Here's a way to show a GIF preloader while an AJAX call is executing. We need to prepare our add and remove preloader functions:

```
function addPreloader() {
  // if the preloader doesn't already exist, add one to the page
  if(!document.querySelector('#preloader')) {
    var preloaderHTML = '<img id="preloader" src="https://goo.gl/cNhyvX" />';
    document.querySelector('body').innerHTML += preloaderHTML;
  }
}

function removePreloader() {
  // select the preloader element
  var preloader = document.querySelector('#preloader');
  // if it exists, remove it from the page
  if(preloader) {
    preloader.remove();
  }
}
```

Now we're going to look at where to use these functions.

```
var request = new XMLHttpRequest();
```

Inside the `onreadystatechange` function you should have an if statement with condition: `request.readyState == 4 && request.status == 200`.

If **true**: the request is finished and response is ready that's where we'll use `removePreloader()`.

Else if **false**: the request is still in progress, in this case we'll run the function `addPreloader()`

```
xmlhttp.onreadystatechange = function() {

  if(request.readyState == 4 && request.status == 200) {
    // the request has come to an end, remove the preloader
    removePreloader();
  } else {
    // the request isn't finished, add the preloader
    addPreloader()
  }

};

xmlhttp.open('GET', your_file.php, true);
xmlhttp.send();
```

Section 36.3: Displaying the top JavaScript questions of the month from Stack Overflow's API

We can make an AJAX request to Stack Exchange's API to retrieve a list of the top JavaScript questions for the month, then present them as a list of links. If the request fails or the returns an API error, our promise error handling displays the error instead.

Version ≥ 6

View live results on HyperWeb.

```
const url =
    'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +
    '&tagged=javascript&sort=month&filter=unsafe&key=gik4BOCMC7J9doavgYteRw((';

fetch(url).then(response => response.json()).then(data => {
  if (data.error_message) {
    throw new Error(data.error_message);
  }

  const list = document.createElement('ol');
  document.body.appendChild(list);

  for (const {title, link} of data.items) {
    const entry = document.createElement('li');
    const hyperlink = document.createElement('a');
    entry.appendChild(hyperlink);
    list.appendChild(entry);

    hyperlink.textContent = title;
    hyperlink.href = link;
  }
}).then(null, error => {
  const message = document.createElement('pre');
```

```
  document.body.appendChild(message);
  message.style.color = 'red';

  message.textContent = String(error);
});
```

Section 36.4: Using GET with parameters

This function runs an AJAX call using GET allowing us to send **parameters** (object) to a **file** (string) and launch a **callback** (function) when the request has been ended.

```
function ajax(file, params, callback) {

  var url = file + '?';

  // loop through object and assemble the url
  var notFirst = false;
  for (var key in params) {
    if (params.hasOwnProperty(key)) {
      url += (notFirst ? '&' : '') + key + "=" + params[key];
    }
    notFirst = true;
  }

  // create a AJAX call with url as parameter
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
      callback(xmlhttp.responseText);
    }
  };
  xmlhttp.open('GET', url, true);
  xmlhttp.send();
}
```

Here's how we use it:

```
ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
  // add here the code to be executed when data comes back to this page
  // for example console.log(response) will show the AJAX response in console
});
```

And the following shows how to retrieve the url parameters in `cars.php`:

```
if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
  // they are set, we can use them !
  $response = 'The color of your car is ' . $_REQUEST['color'] . '. ';
  $response .= 'It is a ' . $_REQUEST['type'] . ' model ' . $_REQUEST['model'] . '!';
  echo $response;
}
```

If you had `console.log(response)` in callback function the result in console would have been:

> The color of your car is purple. It is a Volvo model 300!

Section 36.5: Check if a file exists via a HEAD request

This function executes an AJAX request using the HEAD method allowing us to **check whether a file exists in the directory** given as an argument. It also enables us to **launch a callback for each case** (success, failure).

```
function fileExists(dir, successCallback, errorCallback) {
    var xhttp = new XMLHttpRequest;

    /* Check the status code of the request */
    xhttp.onreadystatechange = function() {
        return (xhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* Open and send the request */
    xhttp.open('head', dir, false);
    xhttp.send();
};
```

Section 36.6: Using GET and no parameters

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //parse the response in xhttp.responseText;
    }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

Version ≥ 6

The `fetch` API is a newer promise-based way to make asynchronous HTTP requests.

```
fetch('/').then(response => response.text()).then(text => {
  console.log("The home page is " + text.length + " characters long.");
});
```

Section 36.7: Listening to AJAX events at a global level

```
// Store a reference to the native method
let open = XMLHttpRequest.prototype.open;

// Overwrite the native method
XMLHttpRequest.prototype.open = function() {
    // Assign an event listener
    this.addEventListener("load", event => console.log(XHR), false);
    // Call the stored reference to the native method
    open.apply(this, arguments);
};
```

Chapter 37: Enumerations

Section 37.1: Enum definition using Object.freeze()

Version ≥ 5.1

JavaScript does not directly support enumerators but the functionality of an enum can be mimicked.

```
// Prevent the enum from being changed
const TestEnum = Object.freeze({
    One:1,
    Two:2,
    Three:3
});
// Define a variable with a value from the enum
var x = TestEnum.Two;
// Prints a value according to the variable's enum value
switch(x) {
    case TestEnum.One:
        console.log("111");
        break;

    case TestEnum.Two:
        console.log("222");
}
```

The above enumeration definition, can also be written as follows:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

After that you can define a variable and print like before.

Section 37.2: Alternate definition

The `Object.freeze()` method is available since version 5.1. For older versions, you can use the following code (note that it also works in versions 5.1 and later):

```
var ColorsEnum = {
    WHITE: 0,
    GRAY: 1,
    BLACK: 2
}
// Define a variable with a value from the enum
var currentColor = ColorsEnum.GRAY;
```

Section 37.3: Printing an enum variable

After defining an enum using any of the above ways and setting a variable, you can print both the variable's value as well as the corresponding name from the enum for the value. Here's an example:

```
// Define the enum
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }
Object.freeze(ColorsEnum);
// Define the variable and assign a value
var color = ColorsEnum.BLACK;
```

```
if(color == ColorsEnum.BLACK) {
   console.log(color);    // This will print "2"
   var ce = ColorsEnum;
   for (var name in ce) {
     if (ce[name] == ce.BLACK)
       console.log(name);    // This will print "BLACK"
   }
}
```

Section 37.4: Implementing Enums Using Symbols

As ES6 introduced **Symbols**, which are both **unique and immutable primitive values** that may be used as the key of an `Object` property, instead of using strings as possible values for an enum, it's possible to use symbols.

```
// Simple symbol
const newSymbol = Symbol();
typeof newSymbol === 'symbol' // true

// A symbol with a label
const anotherSymbol = Symbol("label");

// Each symbol is unique
const yetAnotherSymbol = Symbol("label");
yetAnotherSymbol === anotherSymbol; // false


const Regnum_Animale    = Symbol();
const Regnum_Vegetabile = Symbol();
const Regnum_Lapideum   = Symbol();

function describe(kingdom) {

  switch(kingdom) {

    case Regnum_Animale:
        return "Animal kingdom";
    case Regnum_Vegetabile:
        return "Vegetable kingdom";
    case Regnum_Lapideum:
        return "Mineral kingdom";
  }

}

describe(Regnum_Vegetabile);
// Vegetable kingdom
```

The Symbols in ECMAScript 6 article covers this new primitive type more in detail.

Section 37.5: Automatic Enumeration Value

Version ≥ 5.1

This Example demonstrates how to automatically assign a value to each entry in an enum list. This will prevent two enums from having the same value by mistake. NOTE: Object.freeze browser support

```
var testEnum = function() {
    // Initializes the enumerations
    var enumList = [
```

```
        "One",
        "Two",
        "Three"
    ];
    enumObj = {};
    enumList.forEach((item, index)=>enumObj[item] = index + 1);

    // Do not allow the object to be changed
    Object.freeze(enumObj);
    return enumObj;
}();

console.log(testEnum.One); // 1 will be logged

var x = testEnum.Two;

switch(x) {
    case testEnum.One:
        console.log("111");
        break;

    case testEnum.Two:
        console.log("222"); // 222 will be logged
        break;
}
```

Chapter 38: Map

Parameter	Details
`iterable`	Any iterable object (for example an array) containing `[key, value]` pairs.
`key`	The key of an element.
`value`	The value assigned to the key.
`callback`	Callback function called with three parameters: value, key, and the map.
`thisArg`	Value which will be used as **`this`** when executing `callback`.

Section 38.1: Creating a Map

A Map is a basic mapping of keys to values. Maps are different from objects in that their keys can be anything (primitive values as well as objects), not just strings and symbols. Iteration over Maps is also always done in the order the items were inserted into the Map, whereas the order is undefined when iterating over keys in an object.

To create a Map, use the Map constructor:

```
const map = new Map();
```

It has an optional parameter, which can be any iterable object (for example an array) which contains arrays of two elements – first is the key, the seconds is the value. For example:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);
//                   ^key          ^value          ^key            ^value
```

Section 38.2: Clearing a Map

To remove all elements from a Map, use the `.clear()` method:

```
map.clear();
```

Example:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
console.log(map.get(1)); // undefined
```

Section 38.3: Removing an element from a Map

To remove an element from a map use the `.delete()` method.

```
map.delete(key);
```

Example:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.get(3)); // 4
map.delete(3);
console.log(map.get(3)); // undefined
```

This method returns `true` if the element existed and has been removed, otherwise `false`:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.delete(1)); // true
console.log(map.delete(7)); // false
```

Section 38.4: Checking if a key exists in a Map

To check if a key exists in a Map, use the `.has()` method:

```
map.has(key);
```

Example:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.has(1)); // true
console.log(map.has(2)); // false
```

Section 38.5: Iterating Maps

Map has three methods which returns iterators: `.keys()`, `.values()` and `.entries()`. `.entries()` is the default Map iterator, and contains `[key, value]` pairs.

```
const map = new Map([[1, 2], [3, 4]]);

for (const [key, value] of map) {
  console.log(`key: ${key}, value: ${value}`);
  // logs:
  // key: 1, value: 2
  // key: 3, value: 4
}

for (const key of map.keys()) {
  console.log(key); // logs 1 and 3
}

for (const value of map.values()) {
  console.log(value); // logs 2 and 4
}
```

Map also has `.forEach()` method. The first parameter is a callback function, which will be called for each element in the map, and the second parameter is the value which will be used as `this` when executing the callback function.

The callback function has three arguments: value, key, and the map object.

```
const map = new Map([[1, 2], [3, 4]]);
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));
// logs:
// key: 1, value: 2
// key: 3, value: 4
```

Section 38.6: Getting and setting elements

Use `.get(key)` to get value by key and `.set(key, value)` to assign a value to a key.

If the element with the specified key doesn't exist in the map, `.get()` returns `undefined`.

`.set()` method returns the map object, so you can chain `.set()` calls.

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Section 38.7: Getting the number of elements of a Map

To get the numbers of elements of a Map, use the `.size` property:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

Chapter 39: Timestamps

Section 39.1: High-resolution timestamps

`performance.now()` returns a precise timestamp: The number of milliseconds, including microseconds, since the current web page started to load.

More generally, it returns the time elapsed since the `performanceTiming.navigationStart` event.

```
t = performance.now();
```

For example, in a web browser's main context, `performance.now()` returns `6288.319` if the web page began to load 6288 milliseconds and 319 microseconds ago.

Section 39.2: Get Timestamp in Seconds

To get the timestamp in seconds

```
Math.floor((new Date().getTime()) / 1000)
```

Section 39.3: Low-resolution timestamps

`Date.now()` returns the number of whole milliseconds that have elapsed since 1 January 1970 00:00:00 UTC.

```
t = Date.now();
```

For example, `Date.now()` returns `1461069314` if it was called on 19 April 2016 at 12:35:14 GMT.

Section 39.4: Support for legacy browsers

In older browsers where `Date.now()` is unavailable, use `(new Date()).getTime()` instead:

```
t = (new Date()).getTime();
```

Or, to provide a `Date.now()` function for use in older browsers, use this polyfill:

```
if (!Date.now) {
  Date.now = function now() {
    return new Date().getTime();
  };
}
```

Chapter 40: Unary Operators

Section 40.1: Overview

Unary operators are operators with only one operand. Unary operators are more efficient than standard JavaScript function calls. Additionally, unary operators can not be overridden and therefore their functionality is guaranteed.

The following unary operators are available:

Operator	Operation	Example
`delete`	The delete operator deletes a property from an object.	example
`void`	The void operator discards an expression's return value.	example
`typeof`	The typeof operator determines the type of a given object.	example
`+`	The unary plus operator converts its operand to Number type.	example
`-`	The unary negation operator converts its operand to Number, then negates it.	example
`~`	Bitwise NOT operator.	example
`!`	Logical NOT operator.	example

Section 40.2: The typeof operator

The `typeof` operator returns the data type of the unevaluated operand as a string.

Syntax:

```
typeof operand
```

Returns:

These are the possible return values from `typeof`:

Type	Return value
`Undefined`	`"undefined"`
`Null`	`"object"`
`Boolean`	`"boolean"`
`Number`	`"number"`
`String`	`"string"`
`Symbol` (ES6)	`"symbol"`
`Function` object	`"function"`
`document.all`	`"undefined"`
Host object (provided by the JS environment)	Implementation-dependent
Any other object	`"object"`

The unusual behavior of `document.all` with the `typeof` operator is from its former usage to detect legacy browsers. For more information, see [Why is document.all defined but typeof document.all returns "undefined"?]

Examples:

```
// returns 'number'
typeof 3.14;
typeof Infinity;
typeof NaN;               // "Not-a-Number" is a "number"

// returns 'string'
typeof "";
```

```
typeof "bla";
typeof (typeof 1);        // typeof always returns a string

// returns 'boolean'
typeof true;
typeof false;

// returns 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
typeof undeclaredVariable;
typeof void 0;
typeof document.all       // see above

// returns 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/;           // This is also considered an object
typeof [1, 2, 4];         // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1);     // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;
```

Section 40.3: The delete operator

The `delete` operator deletes a property from an object.

Syntax:

```
delete object.property

delete object['property']
```

Returns:

If deletion is successful, or the property did not exist:

- `true`

If the property to be deleted is an own non-configurable property (can't be deleted):

- `false` in non-strict mode.
- Throws an error in strict mode

Description

The `delete` operator does not directly free memory. It can indirectly free memory if the operation means all references to the property are gone.

`delete` works on an object's properties. If a property with the same name exists on the object's prototype chain, the

property will be inherited from the prototype.
`delete` does not work on variables or function names.

Examples:

```
// Deleting a property
foo = 1;              // a global variable is a property of `window`: `window.foo`
delete foo;           // true
console.log(foo);     // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;           // false
console.log(foo);     // 1 (Not deleted)

// Deleting a function
function foo(){ };
delete foo;           // false
console.log(foo);     // function foo(){ } (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;       // true
console.log(foo);     // Object { } (Deleted bar)

// Deleting a property that does not exist
var foo = { };
delete foo.bar;       // true
console.log(foo);     // Object { } (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;       // false  ()
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

Section 40.4: The unary plus operator (+)

The unary plus (+) precedes its operand *and evaluates* to its operand. It attempts to convert the operand to a number, if it isn't already.

Syntax:

```
+expression
```

Returns:

- a `Number`.

Description

The unary plus (+) operator is the fastest (and preferred) method of converting something into a number.

It can convert:

- string representations of integers (decimal or hexadecimal) and floats.
- booleans: `true`, `false`.
- `null`

Values that can't be converted will evaluate to `NaN`.

Examples:

```
+42           // 42
```

```
+"42"          // 42
+true          // 1
+false         // 0
+null          // 0
+undefined     // NaN
+NaN           // NaN
+"foo"         // NaN
+{}            // NaN
+function(){} // NaN
```

Note that attempting to convert an array can result in unexpected return values.
In the background, arrays are first converted to their string representations:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

The operator then attempts to convert those strings to numbers:

```
+[]             // 0   ( === +'' )
+[1]            // 1   ( === +'1' )
+[1, 2]         // NaN ( === +'1,2' )
```

Section 40.5: The void operator

The **void** operator evaluates the given expression and then returns **undefined**.

Syntax:

```
void expression
```

Returns:

- **undefined**

Description

The **void** operator is often used to obtain the **undefined** primitive value, by means of writing `void 0` or `void(0)`. Note that **void** is an operator, not a function, so `()` is not required.

Usually the result of a **void** expression and **undefined** can be used interchangeably.
However, in older versions of ECMAScript, `window.undefined` could be assigned any value, and it is still possible to use **undefined** as name for function parameters variables inside functions, thus disrupting other code that relies on the value of **undefined**.
void will always yield the *true* **undefined** value though.

`void 0` is also commonly used in code minification as a shorter way of writing **undefined**. In addition, it's probably safer as some other code could've tampered with `window.undefined`.

Examples:

Returning **undefined**:

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Changing the value of `undefined` inside a certain scope:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

Section 40.6: The unary negation operator (-)

The unary negation (-) precedes its operand and negates it, after trying to convert it to number.

Syntax:

```
-expression
```

Returns:

- a `Number`.

Description

The unary negation (-) can convert the same types / values as the unary plus (+) operator can.

Values that can't be converted will evaluate to `NaN` (there is no `-NaN`).

Examples:

```
-42           // -42
-"42"         // -42
-true         // -1
-false        // -0
-null         // -0
-undefined    // NaN
-NaN          // NaN
-"foo"        // NaN
-{}           // NaN
-function(){} // NaN
```

Note that attempting to convert an array can result in unexpected return values.
In the background, arrays are first converted to their string representations:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

The operator then attempts to convert those strings to numbers:

```
-[]           // -0  ( === -'' )
-[1]          // -1  ( === -'1' )
-[1, 2]       // NaN ( === -'1,2' )
```

Section 40.7: The bitwise NOT operator (~)

The bitwise NOT (~) performs a NOT operation on each bit in a value.

Syntax:

```
~expression
```

Returns:

- a `Number`.

Description

The truth table for the NOT operation is:

a	NOT a
0	1
1	0

```
1337  (base 10) = 0000010100111001 (base 2)
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

A bitwise not on a number results in: `-(x + 1)`.

Examples:

value (base 10)	value (base 2)	return (base 2)	return (base 10)
`2`	`00000010`	`11111100`	`-3`
`1`	`00000001`	`11111110`	`-2`
`0`	`00000000`	`11111111`	`-1`
`-1`	`11111111`	`00000000`	`0`
`-2`	`11111110`	`00000001`	`1`
`-3`	`11111100`	`00000010`	`2`

Section 40.8: The logical NOT operator (!)

The logical NOT (`!`) operator performs logical negation on an expression.

Syntax:

```
!expression
```

Returns:

- a `Boolean`.

Description

The logical NOT (`!`) operator performs logical negation on an expression.

Boolean values simply get inverted: `!true === false` and `!false === true`.
Non-boolean values get converted to boolean values first, then are negated.

This means that a double logical NOT (`!!`) can be used to cast any value to a boolean:

```
!!"FooBar" === true
!!1 === true
!!0 === false
```

These are all equal to `!true`:

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

These are all equal to `!false`:

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Examples:

```
!true         // false
!-1           // false
!"-1"         // false
!42           // false
!"42"         // false
!"foo"        // false
!"true"       // false
!"false"      // false
!{}           // false
![]           // false
!function(){} // false

!false        // true
!null         // true
!undefined    // true
!NaN          // true
!0            // true
!""           // true
```

Chapter 41: Generators

Generator functions (defined by the `function*` keyword) run as coroutines, generating a series of values as they're requested through an iterator.

Section 41.1: Generator Functions

A *generator function* is created with a `function*` declaration. When it is called, its body is **not** immediately executed. Instead, it returns a *generator object*, which can be used to "step through" the function's execution.

A `yield` expression inside the function body defines a point at which execution can suspend and resume.

```
function* nums() {
    console.log('starting');  // A
    yield 1;                  // B
    console.log('yielded 1'); // C
    yield 2;                  // D
    console.log('yielded 2'); // E
    yield 3;                  // F
    console.log('yielded 3'); // G
}
var generator = nums(); // Returns the iterator. No code in nums is executed

generator.next();  // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next();  // Executes lines C,D returning { value: 2, done: false }
// console: "yielded 1"
generator.next();  // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next();  // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Early iteration exit

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Throwing an error to generator function

```
function* nums() {
    try {
        yield 1;                 // A
        yield 2;                 // B
        yield 3;                 // C
    } catch (e) {
        console.log(e.message);    // D
    }
}

var generator = nums();

generator.next();  // Executes line A returning { value: 1, done: false }
generator.next();  // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!"));  // Executes line D returning { value: undefined, done: true}
// console: "Error!!"
generator.next();  // no code executed. returns { value: undefined, done: true }
```

Section 41.2: Sending Values to Generator

It is possible to *send* a value to the generator by passing it to the `next()` method.

```
function* summer() {
    let sum = 0, value;
    while (true) {
        // receive sent value
        value = yield;
        if (value === null) break;

        // aggregate values
        sum += value;
    }
    return sum;
}
let generator = summer();

// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();

// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);

// close the generator and collect the result
let sum = generator.next(null).value;  // 111
```

Section 41.3: Delegating to other Generator

From within a generator function, the control can be delegated to another generator function using `yield*`.

```
function* g1() {
  yield 2;
  yield 3;
  yield 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

Section 41.4: Iteration

A generator is *iterable*. It can be looped over with a `for...of` statement, and used in other constructs which depend on the iteration protocol.

```
function* range(n) {
    for (let i = 0; i < n; ++i) {
        yield i;
    }
}

// looping
for (let n of range(10)) {
    // n takes on the values 0, 1, ... 9
}

// spread operator
let nums = [...range(3)];  // [0, 1, 2]
let max = Math.max(...range(100));  // 99
```

Here is another example of use generator to custom iterable object in ES6. Here anonymous generator function `function *` used.

```
let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }
};

for(let p of user){
  console.log( p );
}
```

Section 41.5: Async flow with generators

Generators are functions which are able to pause and then resume execution. This allows to emulate async functions using external libraries, mainly q or co. Basically it allows to write functions that wait for async results in order to go on:

```
function someAsyncResult() {
    return Promise.resolve('newValue')
}

q.spawn(function * () {
    var result = yield someAsyncResult()
    console.log(result) // 'newValue'
})
```

This allows to write async code as if it were synchronous. Moreover, try and catch work over several async blocks. If the promise is rejected, the error is caught by the next catch:

```
function asyncError() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
```

```
            reject(new Error('Something went wrong'))
        }, 100)
    })
}

q.spawn(function * () {
    try {
        var result = yield asyncError()
    } catch (e) {
        console.error(e) // Something went wrong
    }
})
```

Using co would work exactly the same but with `co(function * (){...})` instead of `q.spawn`

Section 41.6: Iterator-Observer interface

A generator is a combination of two things - an `Iterator` and an `Observer`.

Iterator

An iterator is something when invoked returns an `iterable`. An `iterable` is something you can iterate upon. From ES6/ES2015 onwards, all collections (Array, Map, Set, WeakMap, WeakSet) conform to the Iterable contract.

> A generator(iterator) is a producer. In iteration the consumer `PULL`s the value from the producer.

Example:

```
function *gen() { yield 5; yield 6; }
let a = gen();
```

Whenever you call `a.next()`, you're essentially `pull`-ing value from the Iterator and `pause` the execution at `yield`. The next time you call `a.next()`, the execution resumes from the previously paused state.

Observer

A generator is also an observer using which you can send some values back into the generator.

```
function *gen() {
  document.write('<br>observer:', yield 1);
}
var a = gen();
var i = a.next();
while(!i.done) {
  document.write('<br>iterator:', i.value);
  i = a.next(100);
}
```

Here you can see that `yield 1` is used like an expression which evaluates to some value. The value it evaluates to is the value sent as an argument to the `a.next` function call.

So, for the first time `i.value` will be the first value yielded (1), and when continuing the iteration to the next state, we send a value back to the generator using `a.next(100)`.

Doing async with Generators

Generators are widely used with `spawn` (from taskJS or co) function, where the function takes in a generator and allows us to write asynchronous code in a synchronous fashion. This does NOT mean that async code is converted to sync code / executed synchronously. It means that we can write code that looks like `sync` but internally it is still `async`.

> Sync is BLOCKING; Async is WAITING. Writing code that blocks is easy. When PULLing, value appears in the assignment position. When PUSHing, value appears in the argument position of the callback.

When you use iterators, you `PULL` the value from the producer. When you use callbacks, the producer `PUSH`es the value to the argument position of the callback.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Here, you pull the value from `a.next()` and in the second, `v => {...}` is the callback and a value is `PUSH`ed into the argument position `v` of the callback function.

Using this pull-push mechanism, we can write async programming like this,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // wait for 100 ms and send 1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

   // wait for 100 ms and send 2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

So, looking at the above code, we are writing async code that looks like it's `blocking` (the yield statements wait for 100ms and then continue execution), but it's actually `waiting`. The `pause` and `resume` property of generator allows us to do this amazing trick.

How does it work ?

The spawn function uses `yield promise` to PULL the promise state from the generator, waits till the promise is resolved, and PUSHes the resolved value back to the generator so it can consume it.

Use it now

So, with generators and spawn function, you can clean up all your async code in NodeJS to look and feel like it's synchronous. This will make debugging easy. Also the code will look neat.

This feature is coming to future versions of JavaScript - as `async...await`. But you can use them today in ES2015/ES6 using the spawn function defined in the libraries - taskjs, co, or bluebird

Chapter 42: Promises

Section 42.1: Introduction

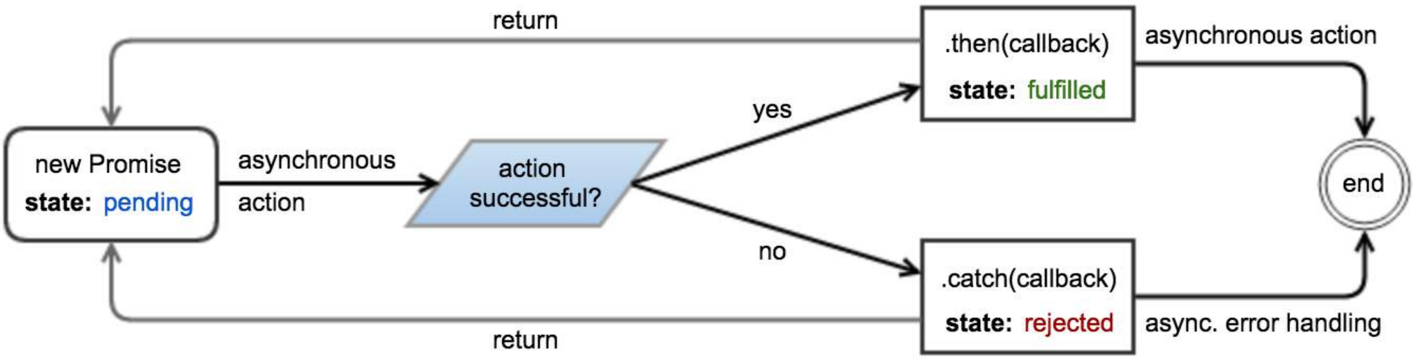
A `Promise` object represents an operation which *has produced or will eventually produce* a value. Promises provide a robust way to wrap the (possibly pending) result of asynchronous work, mitigating the problem of deeply nested callbacks (known as "callback hell").

States and control flow

A promise can be in one of three states:

- *pending* — The underlying operation has not yet completed, and the promise is *pending* fulfillment.
- *fulfilled* — The operation has finished, and the promise is *fulfilled* with a *value*. This is analogous to returning a value from a synchronous function.
- *rejected* — An error has occurred during the operation, and the promise is *rejected* with a *reason*. This is analogous to throwing an error in a synchronous function.

A promise is said to be *settled* (or *resolved*) when it is either fulfilled or rejected. Once a promise is settled, it becomes immutable, and its state cannot change. The `then` and **`catch`** methods of a promise can be used to attach callbacks that execute when it is settled. These callbacks are invoked with the fulfillment value and rejection reason, respectively.



Example

```
const promise = new Promise((resolve, reject) => {
    // Perform some work (possibly asynchronous)
    // ...

    if (/* Work has successfully finished and produced "value" */) {
        resolve(value);
    } else {
        // Something went wrong because of "reason"
        // The reason is traditionally an Error object, although
        // this is not required or enforced.
        let reason = new Error(message);
        reject(reason);

        // Throwing an error also rejects the promise.
        throw reason;
    }
});
```

The `then` and **`catch`** methods can be used to attach fulfillment and rejection callbacks:

```
promise.then(value => {
    // Work has completed successfully,
    // promise has been fulfilled with "value"
}).catch(reason => {
    // Something went wrong,
    // promise has been rejected with "reason"
});
```

Note: Calling `promise.then(...)` and `promise.catch(...)` on the same promise might result in an `Uncaught exception in Promise` if an error occurs, either while executing the promise or inside one of the callbacks, so the preferred way would be to attach the next listener on the promise returned by the previous `then` / `catch`.

Alternatively, both callbacks can be attached in a single call to `then`:

```
promise.then(onFulfilled, onRejected);
```

Attaching callbacks to a promise that has already been settled will immediately place them in the microtask queue, and they will be invoked "as soon as possible" (i.e. immediately after the currently executing script). It is not necessary to check the state of the promise before attaching callbacks, unlike with many other event-emitting implementations.

Live demo

Section 42.2: Promise chaining

The `then` method of a promise returns a new promise.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
    // 5 seconds later
    .then(() => 2)
    // returning a value from a then callback will cause
    // the new promise to resolve with this value
    .then(value => { /* value === 2 */ });
```

Returning a `Promise` from a `then` callback will append it to the promise chain.

```
function wait(millis) {
    return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10 seconds have passed */ });
```

A `catch` allows a rejected promise to recover, similar to how `catch` in a `try`/`catch` statement works. Any chained `then` after a `catch` will execute its resolve handler using the value resolved from the `catch`.

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console));  // outputs "oh yes"
```

If there are no `catch` or `reject` handlers in the middle of the chain, a `catch` at the end will capture any rejection in the chain:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console))      // won't be called
```

```
  .catch(console.error.bind(console));  // outputs "oh yes"
```

On certain occasions, you may want to "branch" the execution of the functions. You can do it by returning different promises from a function depending on the condition. Later in the code, you can merge all of these branches into one to call other functions on them and/or to handle all errors in one place.

```
promise
    .then(result => {
        if (result.condition) {
            return handlerFn1()
                .then(handlerFn2);
        } else if (result.condition2) {
            return handlerFn3()
                .then(handlerFn4);
        } else {
            throw new Error("Invalid result");
        }
    })
    .then(handlerFn5)
    .catch(err => {
        console.error(err);
    });
```

Thus, the execution order of the functions looks like:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~> .catch()
         |                            ^
         V                            |
         -> handlerFn3 -> handlerFn4 -^
```

The single `catch` will get the error on whichever branch it may occur.

Section 42.3: Waiting for multiple concurrent promises

The `Promise.all()` static method accepts an iterable (e.g. an `Array`) of promises and returns a new promise, which resolves when **all** promises in the iterable have resolved, or rejects if **at least one** of the promises in the iterable have rejected.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
]).then(values => console.log(values)); // outputs "[1, 2, 3]" after 7 seconds.

Promise.all([
    resolve(1, 5000),
    reject('Error!', 6000),
    resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
```

```
.catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

Non-promise values in the iterable are "promisified".

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

Destructuring assignment can help to retrieve results from multiple promises.

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
    console.log(result1);
    console.log(result2);
    console.log(result3);
});
```

Section 42.4: Reduce an array to chained promises

This design pattern is useful for generating a sequence of asynchronous actions from a list of elements.

There are two variants :

- the "then" reduction, which builds a chain that continues as long as the chain experiences success.
- the "catch" reduction, which builds a chain that continues as long as the chain experiences error.

The "then" reduction

This variant of the pattern builds a `.then()` chain, and might be used for chaining animations, or making a sequence of dependent HTTP requests.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.then(() => {
        console.log(n);
        return new Promise(res => setTimeout(res, 1000));
    });
}, Promise.resolve()).then(
    () => console.log('done'),
    (e) => console.log(e)
);
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Explanation:

1. We call `.reduce()` on a source array, and provide `Promise.resolve()` as an initial value.
2. Every element reduced will add a `.then()` to the initial value.
3. `reduce()`'s product will be Promise.resolve().then(...).then(...).
4. We manually append a `.then(successHandler, errorHandler)` after the reduce, to execute `successHandler` once all the previous steps have resolved. If any step was to fail, then `errorHandler` would execute.

Note: The "then" reduction is a sequential counterpart of `Promise.all()`.

The "catch" reduction

This variant of the pattern builds a `.catch()` chain and might be used for sequentially probing a set of web servers for some mirrored resource until a working server is found.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.catch(() => {
        console.log(n);
        if(n === working_resource) { // 5 is working
            return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
        } else { // all other values are not working
            return new Promise((resolve, reject) => setTimeout(reject, 1000));
        }
    });
}, Promise.reject()).then(
    (n) => console.log('success at: ' + n),
    () => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Explanation:

1. We call `.reduce()` on a source array, and provide `Promise.reject()` as an initial value.
2. Every element reduced will add a `.catch()` to the initial value.
3. `reduce()`'s product will be `Promise.reject().catch(...).catch(...)`.
4. We manually append `.then(successHandler, errorHandler)` after the reduce, to execute `successHandler` once any of the previous steps has resolved. If all steps were to fail, then `errorHandler` would execute.

Note: The "catch" reduction is a sequential counterpart of `Promise.any()` (as implemented in `bluebird.js`, but not currently in native ECMAScript).

Section 42.5: Waiting for the first of multiple concurrent promises

The `Promise.race()` static method accepts an iterable of Promises and returns a new Promise which resolves or rejects as soon as the **first** of the promises in the iterable has resolved or rejected.

```
// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // outputs "3" after 1 second.
```

```
Promise.race([
    reject(new Error('bad things!'), 1000),
    resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second
```

Section 42.6: "Promisifying" functions with callbacks

Given a function that accepts a Node-style callback,

```
fooFn(options, function callback(err, result) { ... });
```

you can promisify it *(convert it to a promise-based function)* like this:

```
function promiseFooFn(options) {
    return new Promise((resolve, reject) =>
        fooFn(options, (err, result) =>
            // If there's an error, reject; otherwise resolve
            err ? reject(err) : resolve(result)
        )
    );
}
```

This function can then be used as follows:

```
promiseFooFn(options).then(result => {
    // success!
}).catch(err => {
    // error!
});
```

In a more generic way, here's how to promisify any given callback-style function:

```
function promisify(func) {
    return function(...args) {
        return new Promise((resolve, reject) => {
            func(...args, (err, result) => err ? reject(err) : resolve(result));
        });
    }
}
```

This can be used like this:

```
const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));

promisedStat('/foo/bar')
    .then(stat => console.log('STATE', stat))
    .catch(err => console.log('ERROR', err));
```

Section 42.7: Error Handling

Errors thrown from promises are handled by the second parameter (`reject`) passed to `then` or by the handler passed to **`catch`**:

```
throwErrorAsync()
```

```
  .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
  .catch(error => { /* handle error here */ });
```

Chaining

If you have a promise chain then an error will cause `resolve` handlers to be skipped:

```
throwErrorAsync()
  .then(() => { /* never called */ })
  .catch(error => { /* handle error here */ });
```

The same applies to your `then` functions. If a `resolve` handler throws an exception then the next `reject` handler will be invoked:

```
doSomethingAsync()
  .then(result => { throwErrorSync(); })
  .then(() => { /* never called */ })
  .catch(error => { /* handle error from throwErrorSync() */ });
```

An error handler returns a new promise, allowing you to continue a promise chain. The promise returned by the error handler is resolved with the value returned by the handler:

```
throwErrorAsync()
  .catch(error => { /* handle error here */; return result; })
  .then(result => { /* handle result here */ });
```

You can let an error cascade down a promise chain by re-throwing the error:

```
throwErrorAsync()
  .catch(error => {
      /* handle error from throwErrorAsync() */
      throw error;
  })
  .then(() => { /* will not be called if there's an error */ })
  .catch(error => { /* will get called with the same error */ });
```

It is possible to throw an exception that is not handled by the promise by wrapping the `throw` statement inside a `setTimeout` callback:

```
new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});
```

This works because promises cannot handle exceptions thrown asynchronously.

Unhandled rejections

An error will be silently ignored if a promise doesn't have a `catch` block or `reject` handler:

```
throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored
```

To prevent this, always use a `catch` block:

```
throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error*/ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error*/ });
```

Alternatively, subscribe to the `unhandledrejection` event to catch any unhandled rejected promises:

```
window.addEventListener('unhandledrejection', event => {});
```

Some promises can handle their rejection later than their creation time. The `rejectionhandled` event gets fired whenever such a promise is handled:

```
window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');

setTimeout(() => p.catch(console.log), 1000);

// Will print 'unhandled', and after one second 'test' and 'handled'
```

The `event` argument contains information about the rejection. `event.reason` is the error object and `event.promise` is the promise object that caused the event.

In Nodejs the `rejectionhandled` and `unhandledrejection` events are called `rejectionHandled` and `unhandledRejection` on `process`, respectively, and have a different signature:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

The `reason` argument is the error object and the `promise` argument is a reference to the promise object that caused the event to fire.

Usage of these `unhandledrejection` and `rejectionhandled` events should be considered for debugging purposes only. Typically, all promises should handle their rejections.

Note: Currently, only Chrome 49+ and Node.js support `unhandledrejection` and `rejectionhandled` events.

Caveats

Chaining with `fulfill` and `reject`

The `then(fulfill, reject)` function (with both parameters not `null`) has unique and complex behavior, and shouldn't be used unless you know exactly how it works.

The function works as expected if given `null` for one of the inputs:

```
// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)
```

However, it adopts unique behavior when both inputs are given:

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)

// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

The `then(fulfill, reject)` function looks like it is a shortcut for `then(fulfill).catch(reject)`, but it is not, and will cause problems if used interchangeably. One such problem is that the `reject` handler does not handle errors from the `fulfill` handler. Here is what will happen:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }, // error in the fulfill handler
        error => { /* this is not called! */ });
```

The above code will result in a rejected promise because the error is propagated. Compare it to the following code, which results in a fulfilled promise:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }) // error in the fulfill handler
    .catch(error => { /* handle error */ });
```

A similar problem exists when using `then(fulfill, reject)` interchangeably with `catch(reject).then(fulfill)`, except with propagating fulfilled promises instead of rejected promises.

Synchronously throwing from function that should return a promise

Imagine a function like this:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

If such function is used in the **middle** of a promise chain, then apparently there is no problem:

```
makeSomethingAsync().
  .then(() => foo('unexpectedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

However, if the same function is called outside of a promise chain, then the error will not be handled by it and will be thrown to the application:

```
foo('unexpectedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

There are 2 possible workarounds:

Return a rejected promise with the error

Instead of throwing, do as follows:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Wrap your function into a promise chain

Your `throw` statement will be properly caught when it is already inside a promise chain:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Section 42.8: Reconciling synchronous and asynchronous operations

In some cases you may want to wrap a synchronous operation inside a promise to prevent repetition in code branches. Take this example:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

The synchronous and asynchronous branches of the above code can be reconciled by redundantly wrapping the synchronous operation inside a promise:

```
var fetch = result
  ? Promise.resolve(result)
  : fetchResult();

fetch.then(processResult);
```

When caching the result of an asynchronous call, it is preferable to cache the promise rather than the result itself. This ensures that only one asynchronous operation is required to resolve multiple parallel requests.

Care should be taken to invalidate cached values when error conditions are encountered.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
```

```
var cachedPromise;

function fetchResult() {
    if (!cachedPromise) {
        cachedPromise = fetch(planets)
            .catch(function (e) {
                // Invalidate the current result to retry on the next fetch
                cachedPromise = null;
                // re-raise the error to propagate it to callers
                throw e;
            });
    }
    return cachedPromise;
}
```

Section 42.9: Delay function call

The `setTimeout()` method calls a function or evaluates an expression after a specified number of milliseconds. It is also a trivial way to achieve an asynchronous operation.

In this example calling the `wait` function resolves the promise after the time specified as first argument:

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
    console.log('5 seconds have passed...');
});
```

Section 42.10: "Promisifying" values

The `Promise.resolve` static method can be used to wrap values into promises.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

If `value` is already a promise, `Promise.resolve` simply recasts it.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
    // 1 second has passed
    // value === 2
});
```

In fact, `value` can be any "thenable" (object defining a `then` method that works sufficiently like a spec-compliant promise). This allows `Promise.resolve` to convert untrusted 3rd-party objects into trusted 1st-party Promises.

```
let resolved = Promise.resolve({
    then(onResolved) {
        onResolved(2);
    }
});
resolved.then(value => {
```

```
    // immediately invoked
    // value === 2
});
```

The `Promise.reject` static method returns a promise which immediately rejects with the given `reason`.

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
    // immediately invoked
    // reason === "Oops!"
});
```

Section 42.11: Using ES2017 async/await

The same example above, Image loading, can be written using async functions. This also allows using the common **`try`**/**`catch`** method for exception handling.

Note: as of April 2017, the current releases of all browsers but Internet Explorer supports async functions.

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.addEventListener('load', () => resolve(img));
        img.addEventListener('error', () => {
            reject(new Error(`Failed to load ${url}`));
        });
        img.src = url;
    });
}

(async () => {

    // load /image.png and append to #image-holder, otherwise throw error
    try {
        let img = await loadImage('http://example.com/image.png');
        document.getElementById('image-holder').appendChild(img);
    }
    catch (error) {
        console.error(error);
    }

})();
```

Section 42.12: Performing cleanup with finally()

There is currently a proposal (not yet part of the ECMAScript standard) to add a **`finally`** callback to promises that will be executed regardless of whether the promise is fulfilled or rejected. Semantically, this is similar to the **`finally`** clause of the **`try`** block.

You would usually use this functionality for cleanup:

```
var loadingData = true;

fetch('/data')
    .then(result => processData(result.data))
    .catch(error => console.error(error))
    .finally(() => {
        loadingData = false;
```

```
    });
```

It is important to note that the **finally** callback doesn't affect the state of the promise. It doesn't matter what value it returns, the promise stays in the fulfilled/rejected state that it had before. So in the example above the promise will be resolved with the return value of `processData(result.data)` even though the **finally** callback returned **undefined**.

With the standardization process still being in progress, your promises implementation most likely won't support **finally** callbacks out of the box. For synchronous callbacks you can add this functionality with a polyfill however:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function(callback) {
        return this.then(result => {
            callback();
            return result;
        }, error => {
            callback();
            throw error;
        });
    };
}
```

Section 42.13: forEach with promises

It is possible to effectively apply a function (`cb`) which returns a promise to each element of an array, with each element waiting to be processed until the previous element is processed.

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
      return;
    }

    // Process next function. Wrap in `Promise.resolve` in case
    // the function does not return a promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // Chain to finish processing.
    return newPromise.then(nextPromise);
  };

  // Kick off the chain.
  return Promise.resolve().then(nextPromise);
};
```

This can be helpful if you need to efficiently process thousands of items, one at a time. Using a regular **for** loop to create the promises will create them all at once and take up a significant amount of RAM.

Section 42.14: Asynchronous API request

This is an example of a simple `GET` API call wrapped in a promise to take advantage of its asynchronous functionality.

```
var get = function(path) {
```

```
  return new Promise(function(resolve, reject) {
    let request = new XMLHttpRequest();
    request.open('GET', path);
    request.onload = resolve;
    request.onerror = reject;
    request.send();
  });
};
```

More robust error handling can be done using the following `onload` and `onerror` functions.

```
request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // Assuming a successful call returns JSON
      resolve(JSON.parse(request.response));
    } else {
      resolve();
  } else {
    reject({
      'status': this.status,
      'message': request.statusText
    });
  }
};

request.onerror = function() {
  reject({
    'status': this.status,
    'message': request.statusText
  });
};
```

Chapter 43: Set

Parameter	Details
iterable	If an iterable object is passed, all of its elements will be added to the new Set. null is treated as undefined.
value	The value of the element to add to the Set object.
callback	Function to execute for each element.
thisArg	Optional. Value to use as this when executing callback.

The Set object lets you store unique values of any type, whether primitive values or object references.

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur **ONCE**; it is unique in the Set's collection. Distinct values are discriminated using the *SameValueZero* comparison algorithm.

Standard Specification About Set

Section 43.1: Creating a Set

The Set object lets you store unique values of any type, whether primitive values or object references.

You can push items into a set and iterate them similar to a plain JavaScript array, but unlike array, you cannot add a value to a Set if the value already exist in it.

To create a new set:

```
const mySet = new Set();
```

Or you can create a set from any iterable object to give it starting values:

```
const arr = [1,2,3,4,4,5];
const mySet = new Set(arr);
```

In the example above the set content would be `{1, 2, 3, 4, 5}`. Note that the value 4 appears only once, unlike in the original array used to create it.

Section 43.2: Adding a value to a Set

To add a value to a Set, use the `.add()` method:

```
mySet.add(5);
```

If the value already exist in the set it will not be added again, as Sets contain unique values.

Note that the `.add()` method returns the set itself, so you can chain add calls together:

```
mySet.add(1).add(2).add(3);
```

Section 43.3: Removing value from a set

To remove a value from a set, use `.delete()` method:

```
mySet.delete(some_val);
```

This function will return `true` if the value existed in the set and was removed, or `false` otherwise.

Section 43.4: Checking if a value exist in a set

To check if a given value exists in a set, use `.has()` method:

```
mySet.has(someVal);
```

Will return `true` if `someVal` appears in the set, `false` otherwise.

Section 43.5: Clearing a Set

You can remove all the elements in a set using the `.clear()` method:

```
mySet.clear();
```

Section 43.6: Getting set length

You can get the number of elements inside the set using the `.size` property

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

This property, unlike `Array.prototype.length`, is read-only, which means that you can't change it by assigning something to it:

```
mySet.size = 5;
mySet.size; // 4
```

In strict mode it even throws an error:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Section 43.7: Converting Sets to arrays

Sometimes you may need to convert a Set to an array, for example to be able to use `Array.prototype` methods like `.filter()`. In order to do so, use `Array.from()` or `destructuring-assignment`:

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Now you can filter the array to contain only even numbers and convert it back to Set using Set constructor:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

`mySet` now contains only even numbers: