Programming Fundamentals Using JavaScript

Learner eBook

Table of Contents

Welcome to the Academic Institute of Excellence

We pride ourselves on our innovative approach to quality education, excellent service delivery, and a modern outlook to technology. AIE creates, develops, supports, and presents innovative progams to create employable, productive, emotionally intelligent, and skilled learners.

Our learning paths are designed by analysing global skills demands and by providing relevant programmes to meet these demands. Our facilitators are qualified subject matter experts with both practical industry experience and tertiary education experience.

The AIE is a revolutionised family of brands, people, and students. We present our programs in a smarter, more efficient, and cost-effective way by utilising innovation and technology, which results in an expanded reach of our learning interventions. We strive towards excellence and on empowering future generations to become problem solvers, critical thinkers and innovators to create the leaders of tomorrow.

We cultivate excellence.

Our VISION

To deliver demand-driven education, built upon the principal of quality education through innovation and technology.

Getting Started with JavaScript

Module 1

Module Overview

In this module we will look at an introduction to JavaScript

During this Module we will look at the following topics:

- Using console.log()
- Using the DOM API
- Using window.alert()
- Using window.prompt()
- Using window.confirm()
- Using the DOM API (with graphical text: Canvas, SVG, or image file)

Using console.log()

Introduction

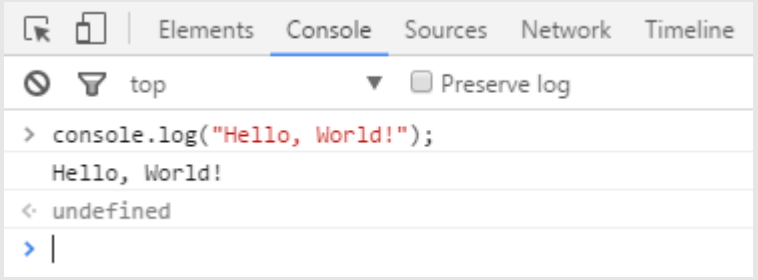
All modern web browsers, Node.js as well as almost every other JavaScript environments support writing messages to a console using a suite of logging methods. The most common of these methods is console.log( ).

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!");
```



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
console.
```

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>
```

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>
```

Take NOTE!

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.

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:

Take NOTE!

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.

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.

Take NOTE!

- 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.

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:

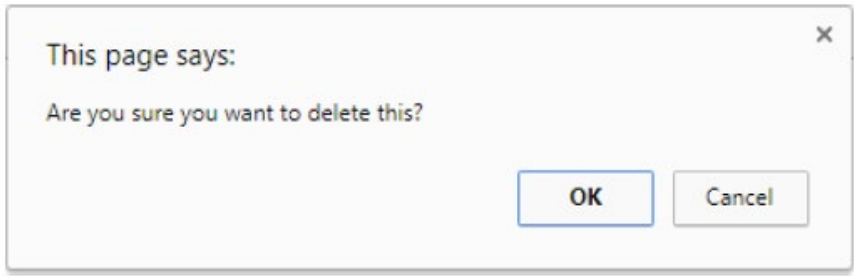
```
result = window.confirm(message);
```

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?");
```

- 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.

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 textelement:

```
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);
```

Comments

Module 2

Module Overview

In this module we will look at how to use comments in JavaScript

During this Module we will look at the following topics:

- Using Comments
- Using HTML comments in JavaScript (Bad practice)

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.cli
}
// 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", elementA
*/
function elementAt( event ) {
  return document.elementFromPoint(event.clientX, event.cli
}
/* TODO: write more useful comments! */
```

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 (<!--):

Take 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).

Variables and constants

Module 3

Module Overview

In this module we will look at variables and Built-in Constants in JavaScript

During this Module we will look at the following topics:

- Defining a Variable
- Using a Variable
- Types of Variables
- Arrays and Objects
- Built-in Constants - Null
- Testing for NaN using isNaN( )
- NaN
- undefined and null
- Infinity and –Infinity
- Number constants
- Operations that return NaN
- Math library functions that return NaN

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.

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.)

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!"
```

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...
```

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

myArray = ["zero", "one", "two"];
window.alert(myArray[0]); // 0 is the first element of an ar
                          // in this case, the value would b
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.

Built-in Constants

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
```

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
```

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
```

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.

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!
```

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;            // 0.00000000000000022204460492503130

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)

Take NOTE!

Number.EPSILON represents the difference 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

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
```

Take NOTE!

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

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)
```

Data Types

Module 4

Module Overview

In this module we will look at Data types and the string function in JavaScript

During this Module we will look at the following topics:

- typeof function
- Finding an object's class
- Getting object type by constructor name
- Basic Info and String Concatenation
- Reverse String
- Comparing Strings Lexicographically
- Access character at index in string
- Escaping quotes
- Word Counter
- Trim whitespace
- Splitting a string into an array
- Strings are Unicode
- Detecting a string
- Substrings with slice
- Character code
- String Representations of Numbers
- String Find and Replace Functions
- Find the index of a substring inside a string
- String to Upper Case
- String to Lower Case
- Repeat a String

Data Types in JavaScript

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"

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
```

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]"

Strings

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

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
```

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

```
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"
```

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" ]
```

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
```

Take NOTE!

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.

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
```

Take 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">.

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`;
```

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

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"
```

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"
```

Strings are unicode

All JavaScript strings are unicode!

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

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

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, );
}
```

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"
```

Character code

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

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

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();
```

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.

```
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); // 0.14158999999999998

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

Take NOTE!

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.

Take NOTE!

Very long representations of numbers may also result in errors due to the accuracy and maximum valuesof *Numbers* of the environment the conversions are happening in.

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.

Take NOTE!

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

Take NOTE!

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!"
```

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
```

String to Upper Case

Take NOTE!

String.prototype.toUpperCase():

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

String to Lower Case

Take NOTE!

String.prototype.toLowerCase()

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

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"
```

Date and Date Comparison

Module 5

Module Overview

In this module we will look at the date and date comparison functions in JavaScript

During this Module we will look at the following topics:

- Create a new Date object
- Convert to a string format
- Creating a Date from UTC
- Formatting a JavaScript date
- Get the number of milliseconds
- Get the current time and date
- Increment a Date Object
- Convert to JSON
- Comparing Date values
- Date Difference Calculation

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.

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 Dateobject.

Take NOTE!

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

Take NOTE!

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
```

```
// 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
```

Convert to a string format

Convert to String

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);
```

Result: "Thursday, April 14, 2016".

Creating a Date from UTC

By default, a Date object is created as local time. This is not always desirable, for example when communicating adate 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);
```

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>

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 24h, 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

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());
```

Get the current time and date

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

Take NOTE!

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
```

Take NOTE!

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());
```

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;
}
```

Convert to JSON

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

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

Date Comparison

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

Take NOTE!

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 isearlier 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

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);
```

Comparison Operations

Module 6

Module Overview

In this module we will look at the comparison operators in JavaScript

During this Module we will look at the following topics:

- Abstract equality / inequality and type conversion
- NaN Property of the Global Object
- Short-circuiting in boolean operators
- Null and Undefined
- Abstract Equality (==)
- Logic Operators with Booleans
- Automatic Type Conversions
- Logic Operators with Non-boolean values (boolean coercion)
- Empty Array
- Equality comparison operations
- Relational operators (<, <=, >, >=)
- Inequality
- List of Comparison Operators
- Grouping multiple logic statements
- Bit fields to optimise comparison of multi state data

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;
```

Take 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 (==)

NaN Property of the Global Object

NaN ("Not a Number") 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 val
```

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

Take NOTE!

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

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
```

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 'prope
if(obj.property && obj !== undefined){}// Line A TypeError:
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 optic
```

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
```

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.

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 beencreated.

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.

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:

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 == '1.00000000001';       // 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
```

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 shortcircuiting 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.

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
```

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*).

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'
```

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.

Take 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'

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
```

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
```

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

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.

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.

Conditions

Module 7

Module Overview

In this module we will look at the conditional expressions in JavaScript

During this Module we will look at the following topics:

- Ternary operators
- Switch statement
- If / Else / If / Else Control
- Strategy
- Using || and && short circuiting

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.

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';
```

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
}
```

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
}
```

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!
```

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.

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")
```

Arrays

Module 8

Module Overview

In this module we will look at the arrays in JS

During this Module we will look at the following topics:

- Converting Array-like Objects to Arrays
- Reducing Values
- Mapping Values
- Filtering Object Arrays
- Sorting Arrays
- Iteration
- Destructuring an array
- Reducing values
- Mapping values
- Filtering Object Arrays
- Sorting Arrays
- Iteration
- Destructuring an array
- Removing duplicate elements
- Array comparison
- Reversing arrays
- Shallow cloning an array
- Concatenating Arrays
- Merge two array as key value pair
- Array spread / rest
- Filtering values
- Searching an Array
- Convert a String to an Array
- Removing items from an array
- Removing all elements
- Finding the minimum or maximum element
- Standard array initialization
- Joining array elements in a string
- Removing/Adding elements using splice()
- The entries() method
- Remove value from array
- Flattening Arrays
- Logical connective of values
- Object keys and values to array
- Insert an item into an array at a specific index
- Checking if an object is an Array
- Insert an item into an array at a specific index
- Sorting multidimensional array
- Test all array items for equality
- Copy part of an Array

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:

```
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:

```
var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}
```

3. Spread operator:

```
[...arrayLike]
```

4. Object.values:

```
var realArray = Object.values(arrayLike);
```

5. Object.keys:

```
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:

```
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:

```
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.

See Arrays are Objects for a detailed analysis.

Reducing values

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]: cur
```

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]
```

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'
});
```

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
}]
```

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.

Take 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']
```

Take 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']
```

Take 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)"
]
```

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:

```
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:

```
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

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.

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])];
```

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**

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]
```

Take 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]
```

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);
```

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"];
```

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"]
```

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"
}
```

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]
```

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"]
```

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
```

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"]
```

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;

Previous
Comparison Operations
Conditions
Arrays
Objects
Arithmetic (Math)
Bitwise Operators
Next >>

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]
```

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:

```
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.

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.

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]
```

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.

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.

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.

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); }
```

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.

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.

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]
]
```

Previous
Comparison Operations
Conditions
Arrays
Objects
Arithmetic (Math)
Bitwise Operators
Next >>

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
```

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
```

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"]
```

```
/* 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"
```

And with array-type arguments merging and chaining support:

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.

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
```

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...]
```

Objects

Module 9

Module Overview

In this module we will look at Objects in JavaScript

During this Module we will look at the following topics:

- Shallow cloning
- Object.freeze
- Object cloning
- Object Properties iteration
- Object.assign
- Object rest/spread (...)
- Object.defineProperty
- Accesor properties (get and set)
- Dynamic / variable property names
- Arrays are Objects
- Object.seal
- Convert object's values to array
- Retrieving properties from an object
- Read-Only property
- Non enumerable property
- Lock property description
- Object.getOwnPropertyDescriptor
- Descriptors and Named Properties
- Object.keys
- Properties with special characters or reserved words
- Creating an Iterable object
- Iterating over Objefct entries- Object.entries()
- Object.values()

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 if assigned to a value

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];
  }
}
```

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';
```

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
```

Take NOTE!

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, []);
}
```

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
```

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" }
```

Take 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})
```

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

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
  }
});
```

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"
```

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;
```

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
```

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
}());
```

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!"]
```

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.

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
```

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
```

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
```

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}
```

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 prop
                  //or calling the set function with the parameter 5 if accessor prope


//*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
```

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"]
```

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.

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
```

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
}
```

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

Arithmetic (Math)

Module 10

Module Overview

In this module we will look at Maths in Javascript

During this Module we will look at the following topics:

- Constants
- Remainder / Modulus (%)
- Rounding
- Trigonometry
- Bitwise operators
- Incrementing (++)
- Exponentiation (Math.pow()or **)
- Random Integers and Floats
- Addition (+)
- Little / big endian for typed arrays when using bitwise operators
- Get Random between two numbers
- Simulating events with di erent probabilities
- Subtraction (-)
- Multiplication (*)
- Getting maximum and minimum
- Restrict number to min/ max range
- Ceiling and Floor
- Getting roots of a number
- Random with Gaussian distribution
- Math.atan2 to find direction
- Sin & Cos to create a vector given direction and distance
- Math.hypot
- Periodic functions using Math.sin
- Division
- Decrementing (--)

Constants

Constants	Description
Approximate	
Math.E 2.718	Base of natural logarithm *e*
Math.LN10 2.302	Natural logarithm of 10
Math.LN2 0.693	Natural logarithm of 2
Math.LOG10E 0.434	Base 10 logarithm of *e*
Math.LOG2E 1.442	Base 2 logarithm of *e*
Math.PI todiameter (π)	Pi: the ratio of circle circumference 3.14
Math.SQRT1_2 0.707	Square root of 1/2
Math.SQRT2 1.414	Square root of 2
Number.EPSILON value greater than one representable as a Number 2.220446049250313080847263336181 6E-16	Difference between one and the smallest
Number.MAX_SAFE_INTEGER 2^53 - 1	Largest integer n such that n and n + 1 are both exactly representable as a Number
Number.MAX_VALUE 1.79E+308	Largest positive finite value of Number
Number.MIN_SAFE_INTEGER both exactly representable as a Number -(2^53 - 1)	Smallest integer n such that n and n – 1 are
Number.MIN_VALUE 5E-324	Smallest positive value for Number
Number.NEGATIVE_INFINITY	Value of negative infinity (-∞)
Number.POSITIVE_INFINITY	Value of positive infinity (∞)
Infinity	Value of positive infinity (∞)

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
```

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
```

ceiling 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
```

flooring 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;
}
```

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.

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*).

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).

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

Module 10 – Arithmetic (Math)

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
```

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

Module 10 – Arithmetic (Math)

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

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"
```

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);
```

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;
```

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
```

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
```

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
```

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
```

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
```

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)

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
```

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
```

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.

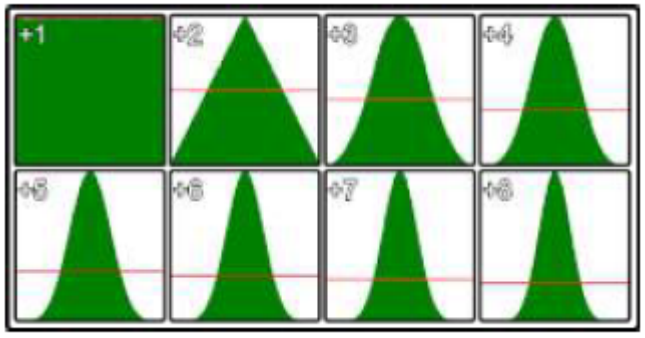
To do this with the Math.random() function is relatively simple.

```
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

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
```

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
```

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

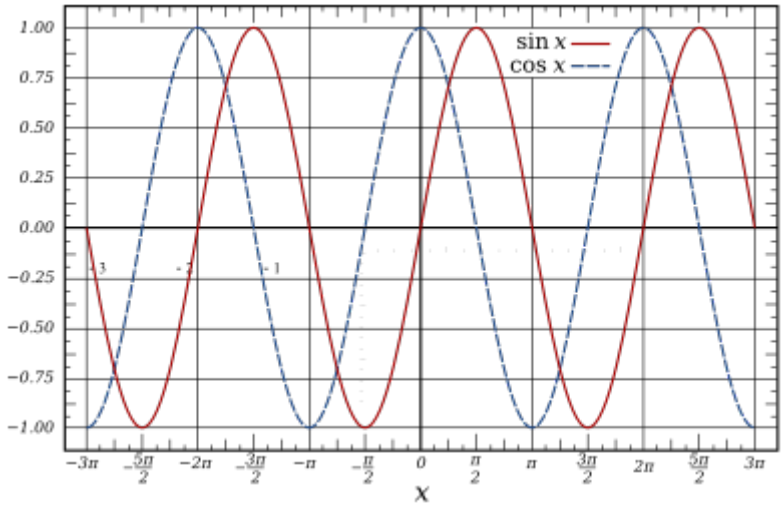
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// 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
```

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 + offs
}
```

All the arguments apart from time are optional

Division (/)

The division operator (/) perform arithmetic division on numbers (literals or variables).

```
console.log(15 / 3); // 5
console.log(15 / 4); // 3.75
```

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:

Previous
Comparison Operations
Conditions
Arrays
Objects
Arithmetic (Math)
Bitwise Operators
Next >>

```
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).

```
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.
```

Take 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.

Bitwise Operators

Module 11

Module Overview

In this module we will look at how Bitwise operators perform in Javascript

During this Module we will look at the following topics:

- Bitwise Operators
- Shift Operators

Bitwise operators

Bitwise operators perform operations on bit values of data. These operators convert operands to signed 32-bit integers in two's 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
```

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.

Previous
Comparison Operations
Conditions
Arrays
Objects
Arithmetic (Math)
Bitwise Operators
Next >>

Constructor Functions

Module 12

Module

In this module we will look at constructor functions that are used to design a new object.

During this Module we will look at the following topics:

- Declaring a constructor function

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"
```

Declarations and Assignments

Module 13

Module Overview

In this module we will look at declarations and assignments in Javascript

During this Module we will look at the following topics:

- Modifying constants
- Declaring and initializing constants
- Declaration
- Undefined
- Data Types
- Mathematic operations and assignment
- Assignment

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.

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.

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.)

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
```

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
```

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;
```

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.

Loops

Module 14

Module Overview

In this module we will look at loops in Javascript

During this Module we will look at the following topics:

- Standard "for" loops
- "for…of" loop
- "for…in" loop
- "while" loops
- "continue" loops
- "continue" a loop
- Break specific nested loops
- "do…while" loop
- Break and continue labels

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

"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.

Take 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
```

"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

"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

"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

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
```

"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.

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
0
1 *i=2 j=2 skips rest of j values*
0
3 1
3 2
i=3 j=3 skips rest of j values
0
4 1
4 2
4 3
i=4 j=4 does not log and loops are done

Functions

Module 15

Module Overview

In this module we will look at functions in Javascript

During this Module we will look at the following topics:

- Higher – order functions
- Identity Monad
- Pure Functions
- Accepting Functions as Arguments

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()();
```

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
```

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.

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]
```

Prototypes, Objects

Module 16

Module

In this module we will look at creating and initialising prototypes

During this Module we will look at the following topics:

- Creation and initialising prototypes

Module 15 – Creation and initialising Prototypes

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)

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();
```

Credits and References

Module 17

About this Book

AIE has reviewed and modified the content of this book for our training purposes from the Javascript® Notes for Professionals book.

This book is compiled from Stack Overflow Documentation and the content is written by the beautiful people at Stack Overflow.

Module 16 - Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content.

2426021684 Chapters 1, 7, 12,
A.M.K Chapters 5, 12
Abdelaziz Mokhnache Chapter 1
afzalex Chapter 3
Ahmed Ayoub Chapter 8
aikeru Chapter 10
Ajedi32 Chapter 13
Ala Eddine JEBALI Chapters 1,
Alberto Nicoletti Chapters 13, 14
Alejandro Nanez Chapter 8
Alex Filatov Chapters 14,
Alex Logan Chapter 5
Alexander O'Mara Chapter 1
Alexandre N. Chapters 1
Aminadav Chapters 1,
Andrew Myers Chapter 3
Andrey Chapter 10
Angelos Chalaris Chapters 13,
Ani Menon Chapters
Anirudh Modi Chapters 12
Anko Chapter 1
Ankur Anand Chapter 1
Ara Yeressian Chapter 3 2
Araknid Chapters 11
AstroCB Chapter 1
Atakan Goktepe Chapter 5
ATechieThought Chapter 1
Ates Goral Chapters 3,
azz Chapter 6
baga Chapter 5
balpha Chapter 8
Bamieh Chapter 8
Barmar Chapter 10
Beau Chapter 5
Bekim Bacaj Chapter 1
Ben Chapter 8
bfavaretto Chapter 1
Black Chapter 1
Blindman67 Chapters 10, 12, 14,
Blubberguy22 Chapter 7
Blue Sheep Chapters 14
BluePill Chapter 7
Blundering Philosopher Chapters 1
bobylito Chapter 3
Borja Tur Chapters 13
Božo Stojković Chapters 1, 12
Brandon Buck Chapter 1
Brett DeWoody Chapter 8
Brett Zamir Chapters 1 and 4
bwegs Chapters 1
C L K Kissane Chapter 5
catalogue_number Chapter 1
cchamberlain Chapter 5
CD.. Chapters 12 and 13
cdrini Chapters 19
Cerbrus Chapters 1, 5, 14, 17
cFreed Chapter 6
Charlie H Chapters 10, 14
Chris Chapters 10
Christian Chapter 3
Christian Landgren Chapter 9
Christoph Chapter 1
Christophe Marois Chapter 3
Claudiu Chapters 7
Cliff Burton Chapters 13
Code Uniquely Chapter 18
codemano Chapter 8
code_monk Chapter 8
CodingIntrigue Chapters 7, 12, 13
Colin Chapter 6
CPHPython Chapters 5, 12
csander Chapters 6, 8, 18
cswl Chapters 15
Daksh Gupta Chapters 1
Damon Chapters 11, 12
Dan Pantry Chapter 3
Daniel Chapter 8
Daniel Herr Chapters 11, 12, 18
daniellmb Chapters 1
dauruy Chapter 8
David Archibald Chapter 1
David G. Chapters 1
Davis Chapters 14,
DawnPaladin Chapters 5
Devid Farinelli Chapters 1
devlin carnate Chapter 3
dns_nx Chapter 8
Domenic Chapters 12
DontVoteMeDown Chapter 1
Dr. J. Testington Chapter 8
Drew Chapter 10
dunnza Chapter 3
DzinX Chapter 8
eltonkamami Chapters 18
Emissary Chapters 5, 17
Erik Minarini Chapter 3
et_l Chapters 13
Evan Bechtol Chapter 3 2
Everettss Chapters 1, 19
fracz Chapters 12
FrankCamara Chapter 8
FredMaggiowski Chapter 9
Gabriel Furstenheim Chapter 1
Gabriel L. Chapter 2
Gaurang Tandon Chapter 10
gca Chapter 6
gcampbell Chapter 7
georg Chapter 3
George Bailey Chapters 12, 13
gman Chapters 1, 5
gnerkus Chapter 7
GOTO 0 Chapters 7
Grundy Chapter 6
H. Pauwelyn Chapters 1

Module 16 - Credits

Hans Strausl Chapters 3
hansmaad Chapter 8
Hardik Kanjariya ツChapters 12, 14
haykam Chapters 1, 5, 7
Hayko Koryun Chapter 10
Hi I'm Frogatto Chapter 7
hiby Chapter 33
hindmost Chapters 14
hirnwunde Chapter 5
hirse Chapter 36
HopeNick Chapters 15
Hunan Rostomyan Chapter 8
Ian Chapters 10
Igor Raush Chapters 10
Inanc Gumus Chapters 1, 5
inetphantom Chapter 1
Ishmael Smyrnow Chapter 8
Isti115 Chapter 8
iulian Chapter 12
J F Chapters 14
James Long Chapter 8
Jamie Chapter 6
Jan Pokorný Chapter 9
Jason Park Chapter 8
JBCP Chapters 3
Jeremy Banks Chapters 1, 10, 12, 13, 14,
Jeremy J Starcher Chapter 8
Jeroen Chapters 1 and 11
jisoo Chapter 8
jitendra varshney Chapter 1
Jivings Chapters 10, jkdev Chapters 3, 10, 12, 18
jmattheis Chapter 1
John Chapter 9
John C Chapter 5
John Slegers Chapters 1, 8, 12,
Jonas W. Chapter 9
Jonathan Lam Chapters 1, 7
Jonathan Walters Chapters 18,
Joshua Kleveter Chapters 1
Just a student Chapters 5
K48 Chapters 1, 9, 10,
kamoroso94 Chapters 8, 14,
Karuppiah Chapter 1
Kevin Katzke Chapter 6
Knu Chapters 10, 11, 13, 14, 18
Kousha Chapter 6
Kyle Blake Chapters 10 and 12
L Bahr Chapters 10
Little Child Chapter 3
little pootis Chapter 1
Louis Barranqueiro Chapters 13
Luís Hendrix Chapters 10
Luc125 Chapters 7 and 12
luisfarzati Chapter 3
M. Erraysy Chapter 8
Maciej Gurban Chapters 12
maheeka Chapter 9
Marco Bonelli Chapters 3
Marco Scabbiolo Chapters 3, 10
Marina K. Chapters 10
Mark Schultheiss Chapters 5
mash Chapter 6
MasterBob Chapters 1,
Matas Vaitkevicius Chapters 1
Mathias Bynens Chapter 1
Mattew Whitt Chapter 3 2
Matthew Crumley Chapters 18
Max Alcala Chapters 12
Maximillian Laumeister Chapter 3
Md. Mahbubul Haque Chapter 9
MEGADEVOPS Chapter 1
MegaTom Chapter 7
Meow Chapters 7, 11, 14
Michał Perłakowski Chapters 1
Michiel Chapter 8
Mike C Chapters 10, 11, 12, 13, 18
Mike McCaughan Chapters 3, 8, 9, 12, 13, 15
Mikhail Chapters 5, 7, 12, 14
Mimouni Chapters 1 and 12
monikapatel Chapter 5
Morteza Tourani Chapter 8
Motocarota Chapter 3 2
Mottie Chapters 10, 12, 14 and 18
n4m31ess_c0d3r Chapter 6
nalply Chapters 10
Naman Sancheti Chapters 1
nasoj1100 Chapter 8
Nathan Tuggy Chapter 7
ndugger Chapters 17
Neal Chapters 12, 13,
Nelson Teixeira Chapter 8
nem035 Chapters 10, 12,
Nhan Chapters 12
ni8mr Chapters 10 and 18
nicael Chapters 11,
Nick Chapter 1
Nina Scholz Chapters 12
nylki Chapter 1
Oriol Chapter 6
Ortomala Lokni Chapter 6
orvi Chapters 1 and 18
Oscar Jara Chapter 6
oztune Chapters 5, 18,
PageYe Chapter 6
patrick96 Chapter 3 2
Paul S. Chapters 7, 10
Pawel Dubiel Chapters 17
pensan Chapter 10
Peter G Chapter 5
Peter LaBanca Chapter 1
Peter Olson Chapter 9
phaistonian Chapter 8
Phil Chapter 9
pinjasaur Chapter 3
Pranav C Balan Chapter 8
pzp Chapters 8
Qianyue Chapters 12
QoP Chapters 12

Module 16 - Credits

Quill Chapters 7
Rafael Dantas Chapter 8
Rajaprabhu Aravindasamy Chapter 9
Rajesh Chapter 6
Rakitić Chapter 1
RamenChef Chapter 10
Raphael Schweikert Chapter 6
Richard Hamilton Chapters 7, 10, 12, 14
riyaz Chapter 3 2
Roamer Chapter 3 2
Rohit Shelhalkar Chapter 5
Roko C. Buljan Chapters 4, 7, 12, 14
rolando Chapters 12, 13, 18
Ronen Ness Chapters 12, 19
ronnyfm Chapter 1
Ruhul Amin Chapter 3
rvighne Chapters 13,
S Willis Chapter 5
sabithpocker Chapter 7
Sandro Chapter 8
SarathChandra Chapter 7
Saroj Sasmal Chapter 1
SeanKendle Chapter 1
SeinopSys Chapters 1
Shrey Gupta Chapter 8
sielakos Chapters 12
Siguza Chapter 3
SirPython Chapter 5
smallmushroom Chapter 18
Spencer Wieczorek Chapters 7, 10, 15
splay Chapters 7,
ssc Chapter 1
stackoverfloweth Chapter 9
Stephen Leppik Chapter 3
Steve Greatrex Chapter 3
Stewartside Chapter 10
still_learning Chapters 14
sudo bangbang Chapter 3
Sumit Chapter 7
Sumurai8 Chapters 8, 10, 13, 14, 17
svarog Chapters 7, 17
Sverri M. Olsen Chapter 1
SZenC Chapters 1, 10, 11, 13, 14, 18
tcooc Chapter 3 2
teppic Chapter 3 2
Thriggle Chapters 1
tjfwalker Chapter 7
tnga Chapter 1
Tolen Chapter 1
Tomás Cañibano Chapters 7
Tomboyo Chapter 20
Traveling Tech Guy Chapter 8
Travis Acton Chapter 1
Trevor Clarke Chapters 8, 14
Tschallacka Chapter 45
Tushar Chapters 1
user2314737 Chapters 8, 12, 14
user3882768 Chapter 7
Vaclav Chapter 8
VahagnNikoghosian Chapters 12
Vasiliy Levykin Chapters 3, 10
Ven Chapters 1, 10
Victor Bjelkholm Chapter 5
VisioN Chapters 12
Vladimir Gabrielyan Chapter 3 2
whales Chapters 8 and 18
Wladimir Palant Chapters 5, 10
wuxiandiejia Chapters 7, 12
XavCo7 Chapters 1, 11, 12, 13, 18
xims Chapter 1
Yosvel Quintero Chapters 1, 5, 12, 13, 14, 17
Yury Fedorov Chapters 1
Zaz Chapter 3 2
zb' Chapter 3 2
zer00ne Chapter 8
ZeroBased_IX Chapter 8
zhirzh Chapters 12, 14
Zoltan.Tamasi Chapter 3
Zze Chapter 1