Grammar and types

JavaScript is case-sensitive and uses the Unicode character set.

Declarations

[var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) - Declares a variable, optionally initializing it to a value.

[let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) - Declares a block scope local variable, optionally initializing it to a value.

[const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) - Declares a read-only named constant.

console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not defined

let y;

The undefined value behaves as false when used in a boolean context.

The undefined value converts to NaN when used in numeric context.

The null value behaves as 0 in numeric contexts and as false in boolean contexts.

Scoping and Hoisting

**Scoping**

In JavaScript, functions are our de facto scope delimiters for declaring vars, which means that usual blocks from loops and conditionals (such as if, for, while, switch and try) DON'T delimit scope, unlike most other languages. Therefore, those **blocks will share the same scope as the function which contains them**. This way, it might be dangerous to declare vars inside blocks as it would seem the var belongs to that block only.

**Hoisting**

On runtime, all **var and function declarations** are **moved to the beginning** of each function (its scope) - this is known as Hoisting. It is a good practice to declare all the vars altogether on the first line, in order to avoid false expectations with a var that got declared late but happened to hold a value before - this is a common problem for programmers coming from languages with block scope.

**Variables** that are hoisted will return a value of **undefined**.

A **constant** cannot change value through assignment or be re-declared while the script is running. But the **properties** of objects assigned to constants are **not** **protected**

For functions, only **function declaration gets hoisted** to the top and not the function expression.

In ECMAScript 2015, **let (const)** will **not hoist** the variable to the top of the block. However, referencing the variable in the block before the variable declaration results in a [ReferenceError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError).

Data structures and types

**Data types**

Primitive

[string](https://developer.mozilla.org/en-US/docs/Glossary/string), [number](https://developer.mozilla.org/en-US/docs/Glossary/number), [boolean](https://developer.mozilla.org/en-US/docs/Glossary/boolean), [null](https://developer.mozilla.org/en-US/docs/Glossary/null), [undefined](https://developer.mozilla.org/en-US/docs/Glossary/undefined), [symbol](https://developer.mozilla.org/en-US/docs/Glossary/symbol) (new in [ECMAScript](https://developer.mozilla.org/en-US/docs/Glossary/ECMAScript) 2015)

Object

**Data type conversion**

Converting strings to numbers

* parseInt(string, radix); // An integer between 2 and 36 that represents the radix (the base in mathematical numeral systems)
* parseFloat(string)

Literals

**Array literals**

If you put two commas in a row, the array is created with undefined for the unspecified elements.

If you include a trailing comma at the end of the list of elements, the comma is ignored.

**Boolean literals**

The Boolean object is a wrapper around the primitive Boolean data type.

**Integers**

Integers can be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2).

|  |  |
| --- | --- |
| Leading | base |
| without a leading 0 | 10(Decimal) |
| 0, 0o (or 0O) | 8(Octal) |
| 0x (or 0X) | 16(Hexadecimal) |
| 0b (or 0B) | 2(Binary) |

**Floating-point literals**

[(+|-)][digits][.digits][(E|e)[(+|-)]digits]

The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit and either a decimal point or "e" (or "E").

**Object literals**

Property names that are not valid identifiers also **cannot** be **accessed** as a **dot** (.) property, but can be **accessed** and set with the array-like **notation**("[]").

var unusualPropertyNames = {  
 '': 'An empty string',  
 '!': 'Bang!'  
}

console.log(unusualPropertyNames.'');   // SyntaxError: Unexpected string  
console.log(unusualPropertyNames['']);  // An empty string  
console.log(unusualPropertyNames.!);    // SyntaxError: Unexpected token !  
console.log(unusualPropertyNames['!']); // Bang!

**RegExp literals**

A regex literal is a pattern enclosed between slashes.

var phoneno = /^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$/;

**String literals**

Using backslash character before special characters to display.

You can also escape line breaks by preceding them with backslash. The backslash and line break are both removed from the value of the string.

Inheritance and the prototype chain

Each object has an internal link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this **prototype chain**.

Inheritance with the prototype chain

**Inheriting properties**

the notation someObject.[[Prototype]] is used to designate the prototype of someObject. This is equivalent to the JavaScript property \_\_proto\_\_ . Since ECMAScript 2015, the [[Prototype]] is accessed using the accessors [Object.getPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf) and [Object.setPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf).

// Let's assume we have object o, with its own properties a and b:  
// {a: 1, b: 2}  
// o.[[Prototype]] has properties b and c:  
// {b: 3, c: 4}  
// Finally, o.[[Prototype]].[[Prototype]] is null. This is the end of the prototype chain, as null,  
// by definition, has no [[Prototype]]. Thus, the full prototype chain looks like:  
// {a: 1, b: 2} ---> {b: 3, c: 4} ---> null

**Inheriting "methods"**

In JavaScript, any function can be added to an object in the form of a property.

When an inherited function is executed, the value of this points to the inheriting object, not to the prototype object where the function is an own property.

var o = {  
 a: 2,  
 m: function() {  
   return this.a + 1;  
 }  
};

var p = Object.create(o); // p is an object that inherits from o

p.a = 4; // creates an own property 'a' on p  
console.log(p.m()); // 5  
// when p.m is called, 'this' refers to p.  
// So when p inherits the function m of o,   
// 'this.a' means p.a, the own property 'a' of p

Create objects and the resulting prototype chain

**Objects created with syntax constructs**

var o = {a: 1}; // o ---> Object.prototype ---> null

var a = ['yo', 'whadup', '?']; // a ---> Array.prototype ---> Object.prototype ---> null

function f() { // f ---> Function.prototype ---> Object.prototype ---> null  
 return 2;  
}

**With a constructor**

A "constructor" in JavaScript is "just" a function to be called with the **new** operator.

function Graph() {  
 this.vertices = [];  
 this.edges = [];  
}  
Graph.prototype = {  
 addVertex: function(v) {  
   this.vertices.push(v);  
 }  
};  
var g = new Graph();  
// g is an object with own properties 'vertices' and 'edges'.  
// g.[[Prototype]] = Graph.prototype when new Graph() is executed.

When access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in [[Prototype]]. This means that all the stuff you define in prototype is effectively shared by all instances, and you can even later change parts of prototype and have the changes appear in all existing instances.

prototype is for types, while Object.getPrototypeOf() is the same for instances.

**With Object.create**

ECMAScript 5 introduced a new method: [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create).

var a = {a: 1}; // a ---> Object.prototype ---> null  
  
var b = Object.create(a); // b ---> a ---> Object.prototype ---> null  
console.log(b.a); // 1 (inherited)  
  
var c = Object.create(b); // c ---> b ---> a ---> Object.prototype ---> null  
  
var d = Object.create(null); // d ---> null  
console.log(d.hasOwnProperty);  // undefined, because d doesn't inherit from Object.prototype

Subclass extends superclass (Inheritance)

// Shape - superclass  
function Shape() {  
 this.x = 0;  
 this.y = 0;  
}  
// superclass method  
Shape.prototype.move = function(x, y) {  
 this.x += x;  
 this.y += y;  
 console.info('Shape moved.');  
};  
// Rectangle - subclass  
function Rectangle() {  
 Shape.call(this); // call super constructor.  
}  
// subclass **extends** superclass  
Rectangle.prototype = Object.create(Shape.prototype); // **Once we override the prototype, we need to consider constructor**  
**Rectangle.prototype.constructor = Rectangle; // inherit Shape’s prototype, including cunstructor**

Create child object

var copyOfParent = Object.create(parentObject.prototype); // copyOfParent object now has everything the parentObject has ​

**With the class keyword**

JavaScript remains prototype-based. The new keywords include [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class), [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor), [static](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static), [extends](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends), and [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super).

'use strict';  
class Polygon {  
 constructor(height, width) {  
   this.height = height;  
   this.width = width;  
 }  
}  
class Square extends Polygon {  
 constructor(sideLength) {  
   super(sideLength, sideLength);  
 }  
 get area() {  
   return this.height \* this.width;  
 }  
 set sideLength(newLength) {  
   this.height = newLength;  
   this.width = newLength;  
 }  
}  
var square = new Square(2);

**Performance**

1. trying to access nonexistent properties will always traverse the full prototype chain.
2. when iterating over the properties of an object, every enumerable property that is on the prototype chain will be enumerated.
3. [hasOwnProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) is the only thing in JavaScript which deals with properties and does not traverse the prototype chain.
4. It is not enough to check whether a property is [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined). The property might very well exist, but its value just happens to be set to undefined.
5. the native prototypes should never be extended unless it is for the sake of compatibility with newer JavaScript features.

**Abstract class**

/\*\* @constructor @abstract \*/

var Animal = function() {

if (this.constructor === Animal) {

throw new Error("Can't instantiate abstract class!");

} // Animal initialization...

};

/\*\* @abstract \*/

Animal.prototype.say = function() {

throw new Error("Abstract method!");

}

var Cat = function() {

Animal.apply(this, arguments); // Cat initialization...

};

Cat.prototype = Object.create(Animal.prototype);

Cat.prototype.constructor = Cat;

Cat.prototype.say = function() { console.log('meow'); }

HTML DOM Elements

var x = document.getElementById("main");

var y = x.getElementsByTagName("p");

var z = document.getElementsByClassName("intro");

window.location.href

get the website URL

location.search

Get from the ? character to the end of the URL or the start of the fragment identifier (#foo), whichever comes first

CSS Selectors

var x = document.querySelectorAll("p.intro");

**NodeList** objects are collections of nodes such as those returned by Node.childNodes and the document.querySelectorAll method.

|  |  |
| --- | --- |
| .item(index) | = nodeList[index]  index is the index of the node to be fetched. The index is zero-based. |
| .forEach(callback); .forEach(callback, argument); | callback - Function to execute for each element, eventually taking 4 arguments (currentValue, currentIndex, listObj, argument)  argument - Value to use as **this** when executing callback. |
| .entries() | iterator allowing to go through all key/value pairs contained in this object. The values are Node objects. |
| .keys() | iterator allowing to go through all keys contained in this object. The keys are unsigned integer. |
| .values() | iterator allowing to go through all values contained in this object. The values are Node objects. |

// Using for..of to go through the iterator  
for(var value of list.values()) {   
 console.log(value);   
}

for...of is a new way for iterating collections.

for...in loops over enumerable property names of an object.

foreach is an method that is available only in Array objects.

HTML Object Collections

**HTMLCollection**

While a NodeList can contain any node type, an HTMLCollection is supposed to only contain Element nodes. An HTMLCollection provides the same methods as a **NodeList** and additionally a method called **namedItem**.

|  |  |
| --- | --- |
| document.getElementsByTagName("td") | $("td") |
| NodeList(||HTMLCollection) | jQuery object |
| method of the DOM interface | presumably jQuery |
| get **live** view on the DOM tree | get snapshots of the DOM tree in the moment the function was called |

var x = document.forms["frm1"]; // frm1 is the form id

var text = "";

var i;

for (i = 0; i < x.length; i++) {

text += x.elements[i].value + "<br>";

}

document.getElementById("demo").innerHTML = text;

**textContent** often has better performance because the text is not parsed as HTML. Moreover, using textContent can **prevent** **XSS** attacks.

1. [document.anchors](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_anchors)
2. [document.body](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_body)
3. [document.documentElement](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_element)
4. [document.embeds](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_embeds)
5. [document.forms](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_forms)
6. [document.head](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_head)
7. [document.images](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_images)
8. [document.links](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_links)
9. [document.scripts](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_scripts)
10. [document.title](http://www.w3schools.com/js/tryit.asp?filename=tryjs_doc_title)

**Style flex Property**

Let all the flexible items be the same length, regardless of its content:

Math

**Math.max()**

Math.max([value1[, value2[, ...]]])

If no arguments are given, the result is -Infinity.

If at least one of arguments cannot be converted to a number, the result is NaN.

**Math.min()**

Math.min([value1[, value2[, ...]]])

If no arguments are given, the result is Infinity.

If at least one of arguments cannot be converted to a number, the result is NaN.

Find the maximum minimum element in a numeric **array**

var min = Math.min.apply(null, arr),

     max = Math.max.apply(null, arr);

var min = Math.min( ...arr ),

     max = Math.max( ...arr );

The **spread syntax**(ECMAScript 6) allows an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) or multiple variables  (for destructuring assignment) are expected.

RegExp

**regexObj.exec(str)**

The exec() method executes a search for a match in a specified string. Returns a result array, or null.

The returned array has the matched text as the first item, and then one item for each capturing parenthesis that matched containing the text that was captured.

If the match fails, the exec() method returns null.

Regular expressions in JavaScript if defined using the /g flag will carry a state across matches, even if they are actually used on different strings (the lastIndex property)

myString = myString.replace(/\D/g,'');

Use the string's [.replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) method with a regex of \D, which is a shorthand character class that matches all non-digits:

**encodeURI** assumes that the input is a complete URI that might have some characters which need encoding in it.

**encodeURIComponent** will encode everything with special meaning, so you use it for components of URIs such as

var world = "A string with symbols & characters that have special meaning?";

var uri = 'http://example.com/foo?hello=' + encodeURIComponent(world);

Closure

A closure is a function having access to the parent scope, even after the parent function has closed. The self-invoking function only runs once, and returns a function expression. It can access the counter in the parent scope.

Closures provide a means of putting function definitions and expressions inside of other functions. A common use would be binding event handler functions so that 'this' refers to the event object. The Module Pattern is the classic example.

a is an object, with a method publicfunction ( a.publicfunction() ) which calls privatefunction, which only exists inside the closure.

a = (function () {

 var privatefunction = function () {

   alert('hello');

 }

 return {

   publicfunction : function () {

     privatefunction();

   }

 }

})();

**Patterns**

Public

function Constructor(...) {

 this.membername = value;

}

Constructor.prototype.membername = value;

Private

function Constructor(...) {

 var that = this;

 var membername = value;

 function membername(...) {...}

}

Note: The function statement function membername(...) {...} is shorthand for

var membername = function membername(...) {...};

Privileged

function Constructor(...) {

 this.membername = function (...) {...};

}

* **private variables** are declared with the 'var' keyword inside the object, and can only be accessed by private functions and privileged methods.
* **private functions** are declared inline inside the object's constructor (or alternatively may be defined via var functionName=function(){...}) and may only be called by privileged methods (including the object's constructor).
* **privileged methods** are declared with this.methodName=function(){...} and may invoked by code external to the object.
* **public properties** are declared with this.variableName and may be read/written from outside the object.
* **public methods** are defined by Classname.prototype.methodName = function(){...} and may be called from outside the object.
* **prototype properties** are defined by Classname.prototype.propertyName = someValue
* **static properties** are defined by Classname.propertyName = someValue

**Anonymous functions**

For single use methods, like when you need to pass a one-liner of code to another function. Or when you want to scope vars via a closure.

**Problem**

closures inside a loop

|  |  |
| --- | --- |
| for(var i = 0; i < 10; i++) {    setTimeout(function() {      console.log(i);      }, 10); } | loop finishes and before setTimeout get the chance to execute. Anonymous functions keep a reference to i by creating a closure. It prints the number 10 ten times |
| for(var i = 0; i < 10; i++) {    setTimeout((function(i) {      console.log(i);    })(i), 10) }  for(var i = 0; i < 10; i++) {     (function(e) {         setTimeout(function() {             console.log(e);         }, 1000);     })(i);  }  for(var i = 0; i < 10; i++) {     setTimeout(function(e) {         console.log(e);     }, 1000, i);  }  for(var i = 0; i < 10; i++) {     setTimeout(console.log.bind(console, i), 1000);  } | avoid closure by create a IIFE (Immediately Invoked Function Expression), it will create its own scope and pass i to the function. In that case i will be a local variable (will not refer to i in the closure) and value of the i in every loop will be preserved. |

**Curry**

Currying is the process of taking a function that accepts some number of arguments, then creating a new version of that function that can accept an "incomplete" argument list.

var currier = function(fn) {

   var args = Array.prototype.slice.call(arguments, 1);

   return function() {

return fn.apply(this, args.concat(

   Array.prototype.slice.call(arguments, 0)));

   };

};

example:

var sequence = function(start, end) {

   var result = [];

   for(var i=start; i<=end; i++) {

       result.push(i);

   }

   return result;

}

var seq5 = currier(sequence, 1);

seq5(5); // [1,2,3,4,5]

var sum = function(a,b,c) {

 return parseInt(a+b+c);

}

var sum1 = currier(sum, 1);

console.log(sum1(5,3)); // don’t work for (5)(3)

function argsArray(argsObject) {

  return Array.prototype.slice.call(argsObject, 0);

}

function curry(f, n) {

  var args = argsArray(arguments);

  if (n === args.length - 2)

      return f.apply(undefined, args.slice(2));

  return function() {

      return curry.apply(undefined, args.concat(argsArray(arguments)));

  };

}

example:

function sum(a, b, c) {

return a + b + c;

}

curry(sum)(1, 2)(3); // 6

//Prototype version

Function.prototype.curry = function() {  
   if (arguments.length<1) {  
       return this; //nothing to curry. return function  
   }  
   var self = this;  
   var args = toArray(arguments);  
   return function() {  
       return self.apply(this, args.concat(toArray(arguments)));  
   }  
}  
  
function toArray(args) {  
   return Array.prototype.slice.call(args);  
}

sum.curry(1, 2)(3); // 6

**Throttle vs Debounce**

<https://blog.coding.net/blog/the-difference-between-throttle-and-debounce-in-underscorejs>

// 控制台记录窗口滚动事件，每250ms最多触发一次  
$(window).scroll( \_.throttle( log, 250 ) );

// 当用户停顿250毫秒以后才开始查找  
$('input:text').keyup( \_.debounce( ajax\_lookup, 250 ) );

// debounce函数用来包裹我们的事件

function debounce(fn, delay) {

// 持久化一个定时器 timer

let timer = null;

// 闭包函数可以访问 timer

return function() {

  // 通过 'this' 和 'arguments'

  // 获得函数的作用域和参数

  let context = this;

  let args = arguments;

  // 如果事件被触发，清除 timer 并重新开始计时

  clearTimeout(timer);

  timer = setTimeout(function() {

    fn.apply(context, args);

  }, delay);

}

}

**JQuery style chaining**

function slow(callback) {

   setTimeout(function(){

       if (Math.random() > 0.5) {

           return callback("Error 417",null)

       }

       callback(null, {id:123})

   },500);

}

// version 1: create promise obj

function exec(fn){

//write your code here

    var promise = {

       callbacks: {

           done: null,

           fail: null

       },

       done: function(cb) {

           this.callbacks.done = cb;

           return this;

       },

       fail: function(cb) {

           this.callbacks.fail = cb;

           return this;

       }

   }

   fn(function(err, result) {

       var cbs = promise.callbacks;

       if(err) return cbs.fail && cbs.fail(err);

       cbs.done && cbs.done(result);

   });

   return promise;

}

//version 2: iQuery

function exec(fn){

//write your code here

 var dfd = $.Deferred();

   var callback = function(fail, result){

       if (fail) {

           dfd.reject(fail)

       }

       if (result){

           dfd.resolve(result);

       }

   };

   fn(callback);

   return dfd;

}

exec(slow).done(function(data){

   console.log(data);

}).fail(function(err){

   console.log("Error: " + err);

});

Array

**Prototype Functions**

shift

removes the first element from an array and returns that element.

var a = [1, 2, 3];  
var b = a.shift();  
console.log(a); // [2, 3]  
console.log(b); // 1

unshift

adds one or more elements to the beginning of an array and returns the new length of the new array.

var a = [1, 2, 3];  
var b = a.unshift(4, 5); // 5  
console.log(a); // [4, 5, 1, 2, 3]

concat

merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

var num1 = [1, 2, 3],  
   num2 = [4, 5, 6],  
   num3 = [7, 8, 9];  
var nums = num1.concat(num2, num3);

Map

map as a "for each" loop, that is specifically for transforming values - **one** input value corresponds **to one** 'transformed' output value.

map is only invoked for elements of the Array which have been initialized.

var numbers = [1, 2, 3, 4];  
var newNumbers = numbers.map(function(number){  
   return number \* 2;  
}).map(function(number){  
   return number + 1;  
});

console.log("The doubled and incremented numbers are", newNumbers); // [3, 5, 7, 9]

['1', '2', '3'].map(parseInt);  
// While one could expect [1, 2, 3]  
// The actual result is [1, NaN, NaN]

['1', '2', '3'].map(Number); // [1, 2, 3]

Filter

The return value from the filter callback should be a **boolean**, indicating whether to include the original value in the result (true) or whether to leave it out (false).

It is not invoked for indexes which have been deleted or which have never been assigned values

var numbers = [1, 2, 3, 4];  
var newNumbers = numbers.filter(function(number){  
   return (number % 2 !== 0);  
}).map(function(number){  
   return number \* 2;  
});  
console.log("The doubled numbers are", newNumbers); // [2, 6]

//find duplicates

var Array1 = ["a", "b", "c", "d", "e", "f"];

var Array2 = ["c", "x", "y", "f"];

Array1 = Array1.filter(function(val) {

 return Array2.indexOf(val) !== -1;

});

// **remove duplicates** in an array

var unique = arr.filter(function(elem, index, self) {

return index == self.indexOf(elem);

})

Reduce

1. The second argument to the function call is considered to be the 'starting value' for the total - this is what you start out with.
2. For each item in the array, it calls the callback, with the total value up to that point, and the item itself. For the first item, the 'total value' is the starting value.
3. You return a new 'total value'. In this case, it's the sum of all previous numbers plus the current number. This return value is used as the 'total value' for the next item.
4. After running out of items, the 'cumulative' total value is returned.

var numbers = [1, 2, 3, 4];  
var newNumbers = numbers.reduce(function(newArray, number){  
   newArray.push(number);  
   if(number % 2 == 0) {  
       /\* Add it a second time. \*/  
       newArray.push(number);  
   }  
   return newArray; /\* This is important! \*/  
}, []);  
console.log("The final numbers are", newNumbers); // [1, 2, 2, 3, 4, 4]

Sum all the values of an array

var sum = [0, 1, 2, 3].reduce(function(a, b) {  
 return a + b;  
}, 0);  
// sum is 6

var total = [ 0, 1, 2, 3 ].reduce( ( acc, cur ) => acc + cur, 0 );

Flatten an array of arrays

var flattened = [[0, 1], [2, 3], [4, 5]].reduce(function(a, b) {  
 return a.concat(b);  
}, []);  
// flattened is [0, 1, 2, 3, 4, 5]

var flattened = [[0, 1], [2, 3], [4, 5]].reduce( ( acc, cur ) => acc.concat(cur), [] );

Counting instances of values in an object

var names = ['Alice', 'Bob', 'Tiff', 'Bruce', 'Alice'];  
var countedNames = names.reduce(function(allNames, name) {   
 if (name in allNames) {  
   allNames[name]++;  
 }  
 else {  
   allNames[name] = 1;  
 }  
 return allNames;  
}, {});  
// countedNames is { 'Alice': 2, 'Bob': 1, 'Tiff': 1, 'Bruce': 1 }

Flatten object keys

var item = {key1: {key2: 2}, key3: {key4:4}};

var res = Object.keys(item).reduce(function(pre,key) {

   for(var sub in item[key]) {

     pre[key+"\_"+sub] = item[key][sub];

   }

 return pre;

}, {});

console.log(res); // {key1\_key2: 2, key3\_key4: 4}

Splice

array.splice(start, deleteCount, item1, item2, ...)

Return value: An array containing the deleted elements. If only one element is removed, an array of one element is returned. If no elements are removed, an empty array is returned.

var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];  
myFish.splice(2, 0, 'drum');   
// myFish is ["angel", "clown", "drum", "mandarin", "sturgeon"]  
myFish.splice(2, 1);   
// myFish is ["angel", "clown", "mandarin", "sturgeon"]

Fill

array.fill(value, start, end)

Return value: The modified array. The elements interval to fill is [start, end).

[1, 2, 3].fill(4);               // [4, 4, 4]  
[1, 2, 3].fill(4, 1);            // [1, 4, 4]  
[1, 2, 3].fill(4, 1, 2);         // [1, 4, 3]

Array(3).fill(4);                // [4, 4, 4]

Slice

var shallowCopy = fruits.slice(); // copy array

// convert Array-like objects / collections to a new Array

function list() {  
 return Array.prototype.slice.call(arguments);  
}  
var list1 = list(1, 2, 3); // [1, 2, 3]

Number

typeof Infinity === 'number';

typeof NaN === 'number';

typeof null === 'object';

When you evaluate a null variable, the null value behaves as 0 in numeric contexts and as false in boolean contexts.

use Math.floor instead of parseInt();

parseInt(1/214748364) = 4

var num = **~~(a / b)**;

var num = **(a / b) >> 0**;

This will work properly for negative numbers as well, while Math.floor() will round in the wrong direction.

The reason >> works is because it operates only on 32-bit integers, so the value is truncated. (It's also commonly used in cases like these instead of Math.floor because bitwise operators have a low operator precedence, so you can avoid a mess of parentheses.

parseInt()

parseInt(string, radix);

The parseInt() function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems). If the first character cannot be converted to a number, [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) is returned.

Because some numbers include the e character in their string representation (e.g. 6.022e23), using parseInt to truncate numeric values will produce unexpected results when used on very large or very small numbers. parseInt should not be used as a substitute for [Math.floor()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/floor).

Floating point precision

0.1+0.2 // 0.30000000000000004

0.1+0.2 == 0.3 // false

way out:

equal = (Math.abs(x - y) < 0.000001)

(0.1 + 0.2).toPrecision(10) == 0.3

(0.1 + 0.2).toFixed(10) == 0.3

isNaN()

isNaN(NaN);       // true  
isNaN(undefined); // true  
isNaN({});        // true  
  
isNaN(true);      // false  
isNaN(null);      // false  
isNaN(37);        // false

// strings  
isNaN('37');      // false: "37" is converted to the number 37 which is not NaN  
isNaN('37.37');   // false: "37.37" is converted to the number 37.37 which is not NaN  
isNaN('123ABC');  // true:  parseInt("123ABC") is 123 but Number("123ABC") is NaN  
isNaN('');        // false: the empty string is converted to 0 which is not NaN  
isNaN(' ');       // false: a string with spaces is converted to 0 which is not NaN  
  
// dates  
isNaN(new Date());                // false  
isNaN(new Date().toString());     // true

function isInteger(x) { return (x^0) === x; }

function isReallyNaN (x){return x!==x;} // fix isNaN(undefined) -> true

ascii to number

'A'.charCodeAt(0)

get a difference in ascii

s.charAt(i).charCodeAt(0)-'A'.charCodeAt(0)

number to ascii

Lowercase: var chr = String.fromCharCode(97 + n);

Uppercase: var chr = String.fromCharCode(65 + n);

null

The value null represents the intentional absence of any object value. It is one of JavaScript's [primitive values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive).

typeof null          // "object" (not "null" for legacy reasons)  
typeof undefined     // "undefined"  
null === undefined   // false  
null  == undefined   // true  
null === null        // true  
null == null         // true  
!null                // true  
isNaN(1 + null)      // false  
isNaN(1 + undefined) // true

Function

When passing in a primitive type variable like a string or a number, the **value is passed in** by value. This means that any changes to that variable while in the function are completely **separate** from anything that happens **outside** the function.

function myfunction(x)  
{  
     // x is equal to 4  
     x = 5;  
     // x is now equal to 5  
}  
  
var x = 4;  // x is equal to 4  
myfunction(x);  // x is still equal to 4

**This**

'this' keyword refers to the object which 'owns' the method, but it depends on how a function is called.

**Bind**

Bind allows you to borrow a method and set the value of **this** without calling the function. It simply returns an exact copy of the function with new value of **this**. You can **reuse** the same function with new value of this without harming the old one. In the new function it is permanently bound to the first argument of bind, bind only works once.

var monica = {  
 name: 'Monica Geller',  
 total: 400,  
 deductMontlyFee: function(fee){  
    this.total = this.total - fee;  
    return this.name + ' remaining balance is '+ this.total;   
 }  
}

var rachel = {name: 'Rachel Green', total: 1500};  
var rachelFeeDeductor = monica.deductMonthlyFee.bind(rachel, 200);  
rachelFeeDeductor(); //"Rachel Green remaining balance is 1300"  
rachelFeeDeductor(); //"Rachel Green remaining balance is 1100"

**Older browser dont have bind function**

Function.prototype.bind = Function.prototype.bind || function(context){  
 var self = this;  
 return function(){  
   return self.apply(context, arguments);  
 };  
}

**Call**

**fun.call(thisArg[, arg1[, arg2[, ...]]])**

A different this object can be assigned when calling an existing function. this refers to the current object, the calling object. With call, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

function Product(name, price) {  
 this.name = name;  
 this.price = price;  
 if (price < 0) {  
   throw RangeError('Cannot create product ' +  
                     this.name + ' with a negative price');  
 }  
}  
function Food(name, price) {  
 Product.call(this, name, price);  
 this.category = 'food';  
}  
function Toy(name, price) {  
 Product.call(this, name, price);  
 this.category = 'toy';  
}  
  
var cheese = new Food('feta', 5);  
var fun = new Toy('robot', 40);

**Apply**

**fun.apply(thisArg, [argsArray])**

The apply() method calls a function with a given this value and arguments provided as an array

Function.prototype.construct = function(aArgs) {  
 var oNew = Object.create(this.prototype);  
 this.apply(oNew, aArgs);  
 return oNew;  
};

function MyConstructor() {  
 for (var nProp = 0; nProp < arguments.length; nProp++) {  
   this['property' + nProp] = arguments[nProp];  
 }  
}  
var myArray = [4, 'Hello world!', false];  
var myInstance = MyConstructor.construct(myArray);  
console.log(myInstance.property1);                // logs 'Hello world!'  
console.log(myInstance instanceof MyConstructor); // logs 'true'  
console.log(myInstance.constructor);              // logs 'MyConstructor'

**Bind vs Call vs Apply**

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

Call/apply call the function immediately, whereas bind returns a function that when later executed will have the correct context set for calling the original function. This way you can maintain context in async callbacks, and events. They both takes the value of **this** as first parameter. We can simply pass **null** as first parameter. However, call takes a collection of arguments after first parameter whereas apply use an array of arguments as second parameter.

String

**slice(), substr() v.s. substring()**

slice() works like substring() with a few different behaviors.

substr() method extracts a specified number of characters in a string, from a start index.

substring() method extracts the chars in a string between two specified indexes.

Syntax: string.slice(start, stop);

Syntax: string.substr(start, length);

Syntax: string.substring(start, stop);

Slice:

* If stop is omitted, slice extracted chars to the end of the string, exactly like substring().
* If start > stop, slice() will NOT swap the 2 arguments. Given empty string
* If start is negative, slice() will set char from the end of string, exactly like substr() in Firefox. This behavior is observed in both Firefox and IE.
* If stop is negative, slice() will set stop to: (string.length – 1) – stop (original value).
* If either argument is NaN, it is treated as if it were 0.

Substr:

* If start is negative, Internet Explorer returns the whole string. That’s wrong! IE should use the last character in the string.
* If either argument is NaN, it is treated as if it were 0.

Substring:

* If start equals stop, it returns an empty string.
* If stop is omitted, it extracts characters to the end of the string.
* If either argument is less than 0 or is NaN, it is treated as if it were 0.
* If either argument is greater than string’s length, either argument will use string’s length.
* If start > stop, then substring will swap those 2 arguments.

**Concat()**

str.concat(string2[, string3, ..., stringN])

Return a new string containing the combined text of the strings provided.

Object

**native object**

object in an ECMAScript implementation whose semantics are fully defined by this specification rather than by the host environment.

NOTE Standard native objects are defined in this specification. Some native objects are built-in; others may be constructed during the course of execution of an ECMAScript program.

Object (constructor), Date, Math, parseInt, eval, string methods like indexOfand replace, array methods, …

**host object**

object supplied by the host environment to complete the execution environment of ECMAScript.

Any object that is not native is a host object.

Host objects (assuming browser environment): window, document, location, history, XMLHttpRequest, setTimeout, getElementsByTagName, querySelectorAll, ...

Time

|  |  |  |  |
| --- | --- | --- | --- |
| setTimeout() | clearTimeout() | setInterval() | clearInterval() |
| var timeoutID = scope.setTimeout(func[, delay, param1, param2, ...]); var timeoutID = scope.setTimeout(code[, delay]); var timeoutID = scope.setTimeout(function, milliseconds); | scope.clearTimeout(timeoutID) | var intervalID = scope.setInterval(func, delay[, param1, param2, ...]); var intervalID = scope.setInterval(code, delay); | scope.clearInterval(intervalID) |
| timeoutID is a numeric, non-zero value | Passing an invalid ID to clearTimeout() silently does nothing; no exception is thrown. | timeoutID is a numeric, non-zero value which identifies the timer created by the call to setInterval() | return undefined |
| Pool of IDs used by setTimeout() and setInterval() are shared. | clearTimeout() and clearInterval() can use interchangeably. But should avoid. |  |  |

AJAX

Asynchronous JavaScript + XML

**Features**

* Make requests to the server without reloading the page
* Receive and work with data from the server

**Advantages**

Better interactivity, AJAX allows easier and quicker interaction between user and website as pages are not reloaded for content to be displayed.

Easier navigation

Compact, several multi purpose applications and features can be handled using a single web page

Backed by reputed brands - Google Maps

**Disadvantages**

The back and refresh button are rendered useless

It is built on javascript, a percentage of website surfers prefer to turn javascript functionality off.

**How AJAX Works**

* 1. An event occurs in a web page (the page is loaded, a button is clicked)
* 2. An XMLHttpRequest object is created by JavaScript
* 3. The XMLHttpRequest object sends a request to a web server
* 4. The server processes the request
* 5. The server sends a response back to the web page
* 6. The response is read by JavaScript
* 7. Proper action (like page update) is performed by JavaScript

**Simple Example**

// Old compatibility code, no longer needed.  
if (window.XMLHttpRequest) { // Mozilla, Safari, IE7+ ...  
   httpRequest = new XMLHttpRequest();  
} else if (window.ActiveXObject) { // IE 6 and older  
   httpRequest = new ActiveXObject("Microsoft.XMLHTTP");  
}

function makeRequest(url) {  
   httpRequest = new XMLHttpRequest();  
   if (!httpRequest) {  
     alert('Giving up :( Cannot create an XMLHTTP instance');  
     return false;  
   }  
   httpRequest.onreadystatechange = alertContents;  
   httpRequest.open('GET', url);  
   httpRequest.send();  
 }  
  
 function alertContents() {  
   if (httpRequest.readyState === 4) {  
     if (httpRequest.status === 200) {  
       alert(httpRequest.responseText); // replace with a callback function  
     } else {  
       alert('There was a problem with the request.');  
     }  
   }  
 }

$.ajax({

       url: 'project.xml',

       dataType: "xml",

       type: "GET",

       success : function(response)

       {

           xml\_file = (new XMLSerializer()).serializeToString(response);

           if (window.DOMParser)

           {

             parser=new DOMParser();

             xmlDoc=parser.parseFromString(xml\_file,"text/xml");

           }

           else // Internet Explorer

           {

             xmlDoc=new ActiveXObject("Microsoft.XMLDOM");

             xmlDoc.async=false;

             xmlDoc.loadXML(xml\_file);

           }

       },

       error: function(XMLHttpRequest, textStatus, errorThrown)

           {

           alert('Data Could Not Be Loaded - '+ textStatus);

       }

   });

**Send a Request To a Server**

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

xhttp.open("POST", "ajax\_test.asp", true);

xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

xhttp.send("fname=Henry&lname=Ford");

|  |  |
| --- | --- |
| Method | Description |
| open(method, url, async) | method: the type of request: GET or POST  url: the server (file) location  async: true (asynchronous) or false (synchronous) |
| send() | Sends the request to the server (used for GET) |
| send(string) | Sends the request to the server (used for POST) |
| setRequestHeader(header, value) | Adds HTTP headers to the request  header: specifies the header name  value: specifies the header value |

GET vs POST

|  |  |  |
| --- | --- | --- |
|  | GET | POST |
| BACK button/Reload | Harmless | Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted) |
| Bookmarked | Can be bookmarked | Cannot be bookmarked |
| Cached | Can be cached | Not cached |
| Encoding type | application/x-www-form-urlencoded | application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data |
| History | Parameters remain in browser history | Parameters are not saved in browser history |
| Restrictions on data length | Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters) | No restrictions |
| Restrictions on data type | Only ASCII characters allowed | No restrictions. Binary data is also allowed |
| Security | GET is less secure compared to POST because data sent is part of the URL  Never use GET when sending passwords or other sensitive information! | POST is a little safer than GET because the parameters are not stored in browser history or in web server logs |
| Visibility | Data is visible to everyone in the URL | Data is not displayed in the URL |

GET is used to retrieve remote data, and POST is used to insert/update remote data.

**Server Response**

|  |  |
| --- | --- |
| Property | Description |
| onreadystatechange | Defines a function to be called every time the readyState property changes |
| readyState | Holds the status of the XMLHttpRequest.  0: request not initialized  1: server connection established  2: request received  3: processing request  4: request finished and response is ready |
| status | 200: "OK"  403: "Forbidden"  404: "Page not found" |
| statusText | Returns the status-text (e.g. "OK" or "Not Found") |

|  |  |
| --- | --- |
| Property | Description |
| responseText | get the response data as a string |
| responseXML | get the response data as XML data (XML DOM object) |

|  |  |
| --- | --- |
| Method | Description |
| getResponseHeader() | Returns specific header information from the server resource |
| getAllResponseHeaders() | Returns all the header information from the server resource |

**JSONP**

JSONP stands for JSON with padding. The padding is a **callback** function that is used to **wrap the data returned from the server**. The reason for its existence is to get around **browser's same-origin restriction** against cross domain requests.

**jQuery AJAX**

Different browsers have different syntax for AJAX implementation. This means that you will have to write extra code to test for different browsers. we can write AJAX functionality with only one single line of code.

The load() method loads data from a server and puts the returned data into the selected element.

$(selector).load(URL,data,callback);

The ajax() method is used to perform an AJAX (asynchronous HTTP) request.

$.ajax({name:value, name:value, ... })

The $.get() method loads data from the server using a HTTP GET request. The GET method may return cached data.

$.get(URL,data,function(data,status,xhr),dataType)

$.get(URL,callback);

The $.post() method loads data from the server using a HTTP POST request. The POST method NEVER caches data, and is often used to send data along with the request.

$(selector).post(URL,data,function(data,status,xhr),dataType)

$.post(URL,data,callback);

Deferred & Promise

* Deffered 触发 resolve 或 reject
* Promise 中申明 resolve 或 reject 后应该做什么（回调）

jQuery

var deferred = $.Deferred();

var promise = deferred.promise();

done() 添加 deferred.resolve() 的回调，fail() 添加 deferred.reject() 的回调。不管成功与否，都会执行 always() 添加的回调。

ES6 Promise

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

   $.ajax().then(resolve, reject);

   // 上面这句没看懂？那换成这样你一定会懂

   // $.ajax().then(function(data) {

       // resolve(data);

   // }, function() {

       // reject();

   // });

}).then(function(data) {

   return data.code ? Promise.reject() : Promise.resolve(data);

   // 这里 Promise.resolve(data) 同样可以直接替换为 data

});

       promise.then(function(data) { // 处理 data

});

Patterns

**module pattern**

a way of organizing and encapsulating code via a closure. It allows you to create **public/private** functions and vars inside an object (the module). It lessens the likelihood of naming conflicts and unintended interactions with other functions/vars on the page. Modules should work independently and be easily extensible. Using modules enables to write widgets and plugins that interact with each other.

var counter = (function(){

var i = 0;

return {

get: function(){

return i;

},

set: function(val){

i = val;

},

increment: function(){

return ++i;

}

}

}());

counter.get();//0

counter.set(3);

counter.increment();//4

counter.increment();//5

conuter.i;//undefined (`i` is not a property of the returned object)

i;//ReferenceError: i is not defined (it only exists inside the closure)

**factory pattern**

function CarFactory() {

   var car = {};

   car.owner = 'nobody';

   var milesPerGallon = 2;

   car.setOwner = function(newOwner) {

       this.owner = newOwner;

   }

   car.getMPG = function() {

       return milesPerGallon;

   }

   return car;

}

While the above is robust against a missing new keyword and certainly makes the use of private variables easier, it comes with some downsides.

1. It uses more memory since the created objects do not share the methods on a prototype.
2. In order to inherit, the factory needs to copy all the methods from another object or put that object on the prototype of the new object.
3. Dropping the prototype chain just because of a left out new keyword is contrary to the spirit of the language.

**Create jQuery API**

$.fn object contains all of the jQuery object methods, and if we want to write our own methods, it will need to contain those as well.

(function($) {  
   $.fn.helloWorld = function( options ) {  
       // Establish our default settings  
       var settings = $.extend({  
           text         : 'Hello, World!',  
           color        : null,  
           fontStyle    : null  
       }, options);  
       return this.each( function() {  
           $(this).text( settings.text );  
           if ( settings.color ) {  
                $(this).css( 'color', settings.color );  
           }  
           if ( settings.fontStyle ) {  
                $(this).css( 'font-style', settings.fontStyle );  
           }  
        });  
      });  
   }  
}(jQuery));

Best Practice

**Optimize Loops**

Don’t make JavaScript read the length of an array at every iteration of a for loop. Store the length value in a different variable.

var names = ['George',   
'Ringo',   
'Paul',   
'John'];  
for(var i=0,j=names.length;i<j;i++){  
  doSomethingWith(names[i]);  
}

**Add Functionality with Javascript Not Content**

It is not convenient to create using the DOM, it’s flasky to use innerHTML (IE’s Operation Aborted error), and it’s hard to keep track of the quality of the HTML you produce.

If you really have a massive interface that should only be available when JavaScript is turned on, load the interface as a static HTML document via Ajax.

**Use === Comparison**

The == comparison operator always converts (to matching types) before comparison.

The === operator forces comparison of values and type

**Use Parameter Defaults**

If a function is called with a missing argument, the value of the missing argument is set to undefined. Undefined values can break your code. It is a good habit to assign default values to arguments.

**Avoid Using eval()**

The eval() function is used to run text as code.

Because it allows arbitrary code to be run, it also represents a security problem.

**Comma operator**

The comma operator evaluates each of its operands (from left to right) and **returns** the value of the **last** operand.

x = (y = 5, z = 6); // Returns 6 in console

return (x += 1, x); // the same as return ++x;

**!!**

Using !! (double bang) is a tricky way to check anything truthy or falsy by avoiding **implicit type conversion** of == comparison.

**== vs ===**

== will not check types and === will check whether both sides are of same type.

implicit conversion

* If both operands are same type use ===
* undefined == null
* If one operands is string another is number, convert string to number
* If one is boolean and another is non-boolean, convert boolean to number and then perform comparison
* While comparing a string or number to an object, try to convert the object to a primitive type and then try to compare

Object Equality

function isEqual(a, b) {  
   var aProps = Object.getOwnPropertyNames(a),  
         bProps = Object.getOwnPropertyNames(b);  
   if (aProps.length != bProps.length) {  
       return false;  
   }  
   for (var i = 0; i < aProps.length; i++) {  
       var propName = aProps[i];  
       if (a[propName] !== b[propName]) {  
           return false;  
       }  
   }  
   return true;  
}

**documentFragment**

DocumentFragment interface represents a minimal document object that has no parent. It is used as a light-weight version of [Document](https://developer.mozilla.org/en-US/docs/Web/API/Document) to store a segment of a document structure comprised of nodes just like a standard document. The key difference is that because the document fragment isn't part of the actual DOM's structure, changes made to the fragment don't affect the document, cause reflow, or incur any performance impact that can occur when changes are made.

var fragment = document.**createDocumentFragment**(),  
   list = ['foo', 'bar', 'baz', ...],  
   el, text;  
for (var i = 0; i < list.length; i++) {  
   el = document.createElement('li');  
   text = document.createTextNode(list[i]);  
   el.appendChild(text);  
   fragment.appendChild(el);  
}  
document.body.appendChild(fragment);

**reflow**

When you change size or position of an element in the page, all the elements after it has to change their position according to the changes you made.

Reasons to reflow:

* change layout (geometry of the page)
* resize the window
* change height/width of any element
* changing font
* change font size
* move DOM element (animation)
* adding or removing stylesheet
* calculating offset height or offset width
* **display: none;**

Avoid:

* avoid setting multiple inline style
* apply animation to the elements that are positioned fixed or absolute
* avoid tables for layout

**Event Bubble**

1. Capture: When you clicked, browser knows a click event occurred. It starts from the window (lowest level/root of your website), then goes to document, then html root tag, then body, then table... its trying to reach the the as lowest level of element as possible. This is called capture phase (phase -1).
2. Target: When browser reach the lowest level of element. In this case, you have clicked on a table cell (table data) hence target would be "td" tag. Then browser checks whether you have any click handler attached to this element. If there is any, browser executes that click hander. This is called target phase (phase -2).
3. Bubbling: After firing click hander attached to "td", browser walks toward root. One level upward and check whether there is any click handler attached with table row ("tr" element). If there is any it will execute that. Then it goes to tbody, table, body, html, document, window. In this stage its moving upward and this is called event bubbling or bubbling phase (phase-3). Please note that, you clicked on cell but all the event handler with parent elements will be fired. This is actually very powerful (check event delegation)

Data Structure

Array

How To

**Scroll to Top Button**

window.onscroll = function() {scrollFunction()};

function scrollFunction() {

   if (document.body.scrollTop > 20 || document.documentElement.scrollTop > 20) {

       document.getElementById("myBtn").style.display = "block";

   } else {

    document.getElementById("myBtn").style.display = "none";

}

}

// When the user clicks on the button, scroll to the top of the document

function topFunction() {

   document.body.scrollTop = 0; // For Chrome, Safari and Opera

   document.documentElement.scrollTop = 0; // For IE and Firefox

}

**CSS Loader**

.loader {

border: 16px solid #f3f3f3; /\* Light grey \*/

border-top: 16px solid #3498db; /\* Blue \*/

border-radius: 50%;

width: 120px;

height: 120px;

animation: spin 2s linear infinite;

}

@keyframes spin {

0% { transform: rotate(0deg); }

100% { transform: rotate(360deg); }

}

**Animated Search Form**

.input[type=text] {

width: 130px;

-webkit-transition: width 0.4s ease-in-out;

transition: width 0.4s ease-in-out;

}

/\* When the input field gets focus, change its width to 100% \*/

input[type=text]:focus {

width: 100%;

}

Linked List

Tree - bt, bst, rbt

Heap

Hash Table

Stack

Queue

Graph

Merge sort

Quick sort

Radix Sort

Data Structure Traversals - bfs,dfs

Algorithm

**Preorder Traversal**

// Iterative

var preorderTraversal = function(root) {

   var res = [];

   var stack = [];

   while(root!==null) {

       res.push(root.val);

       if(root.right!==null){

           stack.push(root.right);

       }

       root = root.left;

       if(root===null && stack.length!==0) {

           root = stack.pop();

       }

   }

   return res;

};

// Recursive

var preorderTraversal = function(root) {

   var res = [];

   helper(res, root);

   return res;

   function helper(res, root) {

       if(root!==null) {

           res.push(root.val);

           if(root.left!==null) {

               helper(res, root.left);

           }

           if(root.right!==null) {

               helper(res, root.right);

           }

       }

   }

};

**Inorder Traversal**

// Iterative

var inorderTraversal = function(root) {

   var stack = [];

   var res = [];

   while(root!==null || stack.length!==0) {

       while(root!==null) {

           stack.push(root);

           root = root.left;

       }

       root = stack.pop();

       res.push(root.val);

       root = root.right;

   }

   return res;

};

// Recursive

var inorderTraversal = function(root) {

   var res = [];

   helper(res, root);

   return res;

   function helper(res, root) {

       if(root!==null) {

           if(root.left!==null) {

               helper(res, root.left);

           }

           res.push(root.val);

           if(root.right!==null) {

               helper(res, root.right);

           }

       }

   }

};

**Postorder Traversal**

// Iterative

var postorderTraversal = function(root) {

   var res = [];

   var stack = [];

   if(root===null) {

       return res;

   }

   stack.push(root);

   while(stack.length!==0) {

       var cur = stack.pop();

       res.unshift(cur.val);

       if(cur.left!==null) {

           stack.push(cur.left);

       }

       if(cur.right!==null) {

           stack.push(cur.right);

       }

   }

   return res;

};

// Recursive

var postorderTraversal = function(root) {

   var res = [];

   helper(res, root);

   return res;

   function helper(res, root) {

       if(root!==null) {

           if(root.left!==null) {

               helper(res, root.left);

           }

           if(root.right!==null) {

               helper(res, root.right);

           }

           res.push(root.val);

       }

   }

};

**Valid BST**

//in-order traversal to serialize a binary search tree

var isValidBST = function(root) {

   var pre\_node = null;

   return isValid(root);

   function isValid(node) {

       if(node) {

           if(!isValid(node.left)) {

               return false;

           }

           if(pre\_node && node.val<=pre\_node.val) {

               return false;

           }

           pre\_node = node;

           return isValid(node.right);

       }

       return true;

   }

};

**Balanced Binary Tree**

var isBalanced = function(root) {

   return depth(root)!==-1;

   function depth(node) {

       if(node===null) {

           return 0;

       }

       var l = depth(node.left);

       if(l===-1) {

           return -1;

       }

       var r = depth(node.right);

       if(r===-1) {

           return -1;

       }

       if(l-r<-1 || l-r>1) {

           return -1;

       }

       return Math.max(l,r)+1;

   }

};

**Binary Tree Level Order Traversal**

var levelOrder = function(root) {

   var res = [];

   var queue = [];

   if(root===null) {

       return res;

   }

   queue.push(root);

   while(queue.length!==0) {

       var len = queue.length;

       var list = [];

       for(var i=0; i<len; i++) {

           var cur = queue.shift();

           list.push(cur.val);

           if(cur.left!==null) {

               queue.push(cur.left);

           }

           if(cur.right!==null) {

               queue.push(cur.right);

           }

       }

       res.push(list); // res.unshift(list) -> bottom up to top order

   }

   return res;

};

BFS vs DFS

|  |  |
| --- | --- |
| BFS | DFS |
| BFS Stands for “Breadth First Search”. | DFS stands for “Depth First Search”. |
| BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node. | DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise. |
| Breadth First Search can be done with the help of queue i.e. FIFO implementation. | Depth First Search can be done with the help of Stack i.e. LIFO implementations. |
| This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once. | This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off. |
| BFS is slower than DFS. | DFS is more faster than BFS. |
| BFS requires more memory compare to DFS. | DFS require less memory compare to BFS. |
| Applications of BFS  > To find Shortest path  > Single Source & All pairs shortest paths  > In Spanning tree  > In Connectivity | Applications of DFS  > Useful in Cycle detection  > In Connectivity testing  > Finding a path between V and W in the graph.  > useful in finding spanning trees & forest. |
| BFS is useful in finding shortest path.BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph. | DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible. |

How to Pick One?

1. Extra Space can be one factor (Explained above)
2. Depth First Traversals are typically recursive and recursive code requires function call overheads.
3. The most important points is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves. So if our problem is to search something that is more likely to closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS.

* [Breadth First Traversal (Or Level Order Traversal)](http://www.geeksforgeeks.org/level-order-tree-traversal/)
* [Depth First Traversals](http://www.geeksforgeeks.org/618/)
  + Inorder Traversal (Left-Root-Right)
  + Preorder Traversal (Root-Left-Right)
  + Postorder Traversal (Left-Right-Root)

**Asynchronous function callback**

var f1 = setTimeout(function() {

   console.log(1);

   var c = call.increase();

   if(c===4) {

     console.log('done');

   }

 },1000);

var f2 = setTimeout(function() {

   console.log(2);

   var c = call.increase();

   if(c===4) {

     console.log('done');

   }

 },2000);

var f3 = setTimeout(function() {

   console.log(3);

   var c = call.increase();

   if(c===4) {

     console.log('done');

   }

 },100);

var f4 = setTimeout(function() {

   console.log(1+3);

   var c = call.increase();

   if(c===4) {

     console.log('done');

   }

 },1000);

var call = (function() {

 var timer = 0;

 return {

   increase: function() {

     timer++;

     return timer;

   }

 }

}());

**Count and say**

var countAndSay = function(n) {

   var res = '1';

   var buffer;

   var count;

   for(var i=1; i<n; i++) {

       buffer = '';

       count = 1;

       var c = res[0];

       for(var j=1; j<res.length; j++) {

           if(res[j]===c) {

               count++;

               continue;

           }

           buffer += count+c;

           count = 1;

           c = res[j];

       }

       buffer += count+c;

       res = buffer;

   }

   return res;

};

**Lowest Common Ancestor of a Binary Search Tree**

var lowestCommonAncestor = function(root, p, q) {

   while((root.val-p.val)\*(root.val-q.val)>0) {

       root = root.val>p.val? root.left : root.right;

   }

   return root;

};

**Lowest Common Ancestor of a Binary Tree**

var lowestCommonAncestor = function(root, p, q) {

   if(root===null || p===root || q===root) {

       return root;

   }

   var left = lowestCommonAncestor(root.left, p, q);

   var right = lowestCommonAncestor(root.right, p, q);

   if(left!==null && right!==null) {

       return root;

   }

   return left!==null ? left:right;

};

**Roman to Integer**

var romanToInt = function(s) {

   var num = [];

   var res = 0;

   for(var i=0; i<s.length; i++) {

       switch(s[i]) {

           case "I":

               num[i] = 1;

               break;

           case "V":

               num[i] = 5;

               break;

           case "X":

               num[i] = 10;

               break;

           case "L":

               num[i] = 50;

               break;

           case "C":

               num[i] = 100;

               break;

           case "D":

               num[i] = 500;

               break;

           case "M":

               num[i] = 1000;

               break;

           default:

               break;

       }

   }

   for(var j=0; j<s.length; j++) {

       if(num[j]<num[j+1]) {

           res -= num[j];

       }else {

           res += num[j];

       }

   }

   return res;

};

**Integer to Roman**

var intToRoman = function(num) {

   var m = ['','M','MM','MMM'];

   var c = ['','C','CC','CCC','CD','D','DC','DCC','DCCC','CM'];

   var x = ['','X','XX','XXX','XL','L','LX','LXX','LXXX','XC'];

   var i = ['','I','II','III','IV','V','VI','VII','VIII','IX'];

   var res = '';

   res += m[Math.floor(num/1000)];

   res += c[Math.floor((num%1000)/100)];

   res += x[Math.floor((num%100)/10)];

   res += i[Math.floor(num%10)];

   return res;

};

**Valid Palindrome**

var isPalindrome = function(s) {

   s = s.replace(/[^\w\d]/g, '');

   if(s===null) return true;

   return s.toUpperCase() === s.split('').reverse().join('').toUpperCase();

}

<h2>Tabs Spec</h2>

<a target="\_blank" href="https://imgur.com/iZtvU5f">Click Here to See Demo</a>

<!-- CODE BELOW HERE -->

<h2>Implementation</h2>

<a href="#boston\_content" data-id="boston\_a" class="tab">Boston</a>

<a href="#paris\_content" data-id="paris\_a" class="tab">Paris</a>

<a href="#tokyo\_content" data-id="tokyo\_a" class="tab">Tokyo</a>

<div data-id="boston\_content" data-place="boston" class="content active">A city in eastern Massachusetts, the capital of the state, on Massachusetts Bay; pop. 609,023 (est. 2008). It was founded c.1630 by the Massachusetts Bay Company under its governor, John Winthrop(1588–1649). Boston was the scene of many disturbances that

 led to the American Revolution at the end of the 18th century.</div>

<div data-id="paris\_content" data-place="paris" class="content">The capital of France, on the Seine River; pop. 2,203,817 (2006). Paris was held by the Romans, who called it Lutetia, and by the Franks, and was established as the capital in 987 under Hugh Capet. It was organized into three parts—the Île de la Cité

 (an island in the Seine), the Right Bank, and the Left Bank—during the reign of Philippe-Auguste 1180–1223.</div>

<div data-id="tokyo\_content" data-place="tokyo" class="content">The capital of Japan, located on the northwestern shores of Tokyo Bay, on the southeastern part of the island of Honshu; pop. 12,758,000 (est. 2007). Formerly called Edo, it was the center of the military government under the shoguns 1603–1867. Renamed

 Tokyo in 1868, it replaced Kyoto as the imperial capital.</div>

// Feel free to configure any libraries you want to use by clicking the gear icon to the left of "JS", and then clicking on the "Quick Add" dropdown

// Look up any documentation you need

// It's OK to use native Javascript APIs, too!

$('a').click(function(e) {

 var curr\_target = e.target;

 var e\_id = $(curr\_target).data('id');

 var place = e\_id.split('\_')[0];

 removeAll();

 if(!$(curr\_target).hasClass('active\_a')) {

       $(curr\_target).addClass('active\_a');

     }

 var content = $(".content[data-place='"+place+"']");

     console.log(content);

     if(!$(content).hasClass('active')) {

       $(content).addClass('active');

     }

});

function removeAll() {

 //remove active class for all

   var content = $(".content");

   $.each(content, function() {

     if($(this).hasClass('active')) {

       $(this).removeClass('active');

     }

   }) ;

   var anchors = $(".tab");

   $.each(anchors, function() {

     if($(this).hasClass('active\_a')) {

       $(this).removeClass('active\_a');

     }

   }) ;

}

Memory

Memory Life cycle

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

Garbage collection

**Reference-counting garbage collection**

Cycles are a common mistake that can generate memory leaks:

var div;  
window.onload = function() {  
 div = document.getElementById('myDivElement');  
 div.circularReference = div;  
 div.lotsOfData = new Array(10000).join('\*');  
};

**Mark-and-sweep algorithm**

This algorithm reduces the definition of "an object is not needed anymore" to "an object is unreachable".