## 📄 About This Document

the purpose of this cheat sheet is to briefly describe the core elements of the JavaScript language for those of studying it who have taken in much more than we can hold onto well. nothing here is explained *in full* but rather meant to get you on the right track. also, this document purposely does not cover browser-specific methods / syntax / objects and the like.

📌 *this cheat sheet is a work in progress and may be updated -- check back on occasion!*

## ⇄ Types

| Type | typeOf evaluation | Primitive? |
|---|---|---|
| Null | object | yes |
| Undefined | undefined | yes |
| Boolean | boolean | yes |
| String | string | yes |
| Number | number | yes |
| Object | object | no --*an object* |
| Function | function | no --*an object* |
| Array | object | no --*an object* |
| Symbol | symbol | no --*an object* |
| [] | object | no --*an object* |
| {} | object | no --*an object* |

## 📊 Number & Math Methods

» `someNum.toFixed(num)`

• shortens someNum to have only num decimal places

» `num.toExponential()`

• converts num to exponential notation (i.e. 5.569e+0)

» `num.toString()`

• converts num to a string

» `num.toPrecision(#)`

• converts num to a num with # places starting with whole numbers

» `String(someValue)`

• converts or coerces someValue to a string - someValue can be any type, ie "Boolean(1)" returns true

» `parseInt(string, radix)`

» `parseFloat(string, radix)`

• converts a string into an integer. the optional radix argument defines the base -- i.e., base 10 (decimal) or base 16 (hexadecimal).

» `Math.round(num)`

• rounds num to nearest integer

## 📊 Number & Math Methods (cont)

» `Math.ceil(num)`

• rounds num *up* to nearest integer

» `Math.floor(num)`

• rounds num *down* to nearest integer

» `Math.max(num1, num2)`

• returns larger num

» `Math.min(num1, num2)`

» `Math.pow(num1, num2)`

• returns num1 to the power num2

» `Math.sqrt(num)`

» `Math.random()`

• returns decimal between 0 (inclusive) and 1(exclusive)

» `Math.abs(num)`

• returns absolute value of num•

## 📢 Array "Extra Methods"

💡 **Note:** these "extra methods," which are "higher-order" functions, ignore holes in the array (i.e.: ["apples", , , , "oranges"]). they also have more arguments than shown here -- best to look them up for more info!

💡 **Note:** array-*like* objects, for example `arguments` and `NodeLists`, can also make use of these methods.

» `arr.some(callback)`

» `arr.every(callback)`

• returns a boolean value. returns true if*some* or *every* element in the array meets the evaluation. example:

```
var a = [1,2,3];
var b = a.every(function(item){
  return item > 1;
}); // false
```

» `arr.reduce(function(prev, next){..}, startVal)`

» `arr.reduceRight(function(prev, next){..}, startVal)`

• returns a value. *reduce* employs a callback to run through the elements of the array, returning "prev" to itself with each iteration and taking the next "next" value in the array. for it's first "prev" value it will take an optional "startVal" if supplied. an interesting example:

```
var arr = ["apple", "pear", "apple", "lemon"];
var c = arr.reduce(function(prev, next) {
 prev[next] = (prev[next] += 1) || 1;
 return prev;
```

## 📢 Array "Extra Methods" (cont)

```
 }, {});
 // objCount = { apple: 2, pear: 1, lemon: 1 }
```

» `arr.filter(function(){..})`

• returns an array. *filter* returns an array of elements that satisfy a given callback. example:

```
 var arr2 = ["jim", "nancy", "ned"];
 var letter3 = arr2.filter(function(item) {
  return (item.length === 3);
 });
 console.log(letter3); // ['jim', 'ned']
```

» `arr.sort(function(){..})`

• returns *the original* array, mutated. *sort* returns the elements sorted with a given criteria. for example:

```
 var stock = [{key: "r", num: 12}, {key: "a", num:
2}, {key: "c", num: 5}];
 var c = stock.sort(function(a,b) {
  return a.num - b.num;
 } ); // [ { key: 'a', num: 2 }, { key: 'c', num: 5
}, { key: 'r', num: 12 } ]
```

» `arr.map()`

• returns an array. *map* goes over every element in the array, calls a callback on the element, and sets an element in the new array to be equal to the return value the callback. for example:

```
 var stock = [{key: "red", num: 12}, {key: "blue",
num: 2}, {key: "black", num: 2}];
 var b = stock.map(function (item){
  return item.key;
 }) // ["red","blue","black"]
```

» `arr.forEach()`

• no return value. *forEach* performs an operation on all elements of the array. for example:

```
 var arr = ["jim", "mary"];
 a.forEach (function (item) {
   console.log("I simply love " +item);
 }); // "I simply love jim", "I simply love mary"
```

💡 **Note:** you can combine array methods in a *chain* where the result of the leftmost operation is passed to the right as such:

```
array.sort().reverse()...
```

## 🚀 Functions & Etc.

💡 **Callbacks:** placing ( ) after a function call *executes it immediately*. leaving these off allows for a callback.

**Function Declaration**

» `function aFunctionName (args) {...`

• functions created in this manner are evaluated when the code is parsed and are 'hoisted' to the top and are available to the code *even before* they're formally declared. Note: Due to JS's odd construction, using function declarations within a flow control statement can be wonky and is best avoided.

**Function Expression / Anonymous Functions**

» `var bar = function (args) {...`

• (also referred to as 'Function Operators') anonymous functions are evaluated at 'runtime' and are therefore less memory intensive. they must be provided a variable name but need not have a function name (therefore: anonymous). [these are ]

**Named Function Expression**

» `var bar = function foo (args) {...`

• confusingly, this is still an 'anonymous function.' assigning a name is useful for debugging purposes and also allows for self-referential / recursive calls

**Function Constructor**

» `var anotherFunction = new Function (args, function`
`() {... }) {...}`

• equivalent to a functional expression

**Self-Invoking Anonymous Functions**

» `( function (args) { doSomething; } ) ( );`

• (also known as IIFEs / 'Immediately Invoked Function Expressions') and invokes immediately

## ↺ Loops / Control Flow Statements

**if .. else if .. else**

```
 if (considtion1) {
   doSomething;
 } else if {
   doSomethingElse;
 } else {
   doSomethingMore;
 }
```

**for loop**

```
 for (var i = 0; i < someNumber; i++) {
   doSomething;
 }
```

**switch loop**

By **AC Winter** (acwinter)

cheatography.com/acwinter/

Published 6th May, 2015.
Last updated 13th May, 2015.
Page 2 of 5.

## ↺ Loops / Control Flow Statements (cont)

```
switch (someEvaluation) {
 case "evaluatesAsThis" :
   doSomething;
 case "evaluatesAsThat" :
   doSomethingElse;
}
```

**while loop**

```
while (someEvaluation === true) {
  doSomething;
}
```

**do .. while**

```
do {
  doSomething;
}
while (someEvaluation === true);
```

**for .. in (objects)**

```
for (anItem in anObject) {
  doSomething With anItem;
   // will be the key
  doSomethingWith Object[anItem];
   // will be the value of that key
}
```

## ⟳ "this"

*coming soon*

## 💬 String Methods, Properties & Etc

💡 a string *can be coerced into an array* so many array methods are applicable as well

» `str.charAt(num)`

• returns the character in str at index num

» `str.charCodeAt(num)`

• returns the unicode value of the char
String.fromCharCode(num)`

• returns the character with unicode's num

» `str.indexOf(char)`

• returns -1 if char not found in str

» `str.lastIndexOf(subString)`

## 💬 String Methods, Properties & Etc (cont)

• returns the index of the last occurrence of subString

» `str.length`

• returns length of str starting at 1

» `str.match(pattern)`

• returns null if not found. returns an *array* of all matches

» `str.match(/pattern/g)`

• provides global search of string

» `str.replace(old, new)`

» `str.search(pattern)`

• returns index of first match or -1 if not found

» `str.substring(index1, index2)`

• char at index1 is returned, index2 is not

» `str.split(char)`

• returns an array of str split on char

» `str.substr(index1, num)`

• returns substring starting at index1 and running num letters

» `str.toLowerCase()`

» `str.toUpperCase()`

» `str.toLocaleLowerCase()`

• takes local language settings into account

» `str.toLocaleUpperCase()`

• ibid

» `Number(var/string/object)`

• converts to number. "true" converts to 1, etc

» `one.concat(two)`

• concatenates string/array one with two

» `JSON.stringify( )`

• converts a javascript value/object into a string

» `JSON.parse ( )`

• converts a JSON string into a javascript object

## 📅 Date Methods

💡 **Note:** Unix epoch is January 1, 1970

» `var today = new Date();`

• creates date object for now

» `var someDate = new Date("june 30, 2035");`

• creates date object for arbitrary date

» `var today = Date.now();`

## 📅 Date Methods (cont)

• returns number of milliseconds since epoch

» `parse()`

• returns milliseconds between date and Unix epoch.

» `toDateString()`

» `toTimeString()`

» `toLocalTimeString()`

## 🕐 Get / Set Date Methods

| | |
|---|---|
| • getDate() | • getHours() |
| • getDay() | • getMilliseconds() |
| • getFullYear() | • getMinutes() |
| • getMonth() | • getSeconds() |
| • getTime() | • getTimezoneOffset() |

💡 **Note:** there are also 'set' methods such as *setMonth().*

💡 **Note:** getDay and getMonth return numeric representations starting with 0.

## 🖥 Miscellaneous Instructions

» `break;`

• breaks out of the current loop

» `continue;`

• stops current loop iteration and increments to next

» `isNaN(someVar)`

• returns true if not a number

» `isFinite(someVar)`

» `var aVar = anObject[anAttribute] || "nonesuch";`

• assigns a default value if none exists

» `var aVar = anEvaluation ? trueVal : falseVal;`

• ternary operator. assigns trueVal to aVar if anEvaluation is true, falseVal if not

» `delete anObject[anAttribute]`

» `(aProperty in anObject)`

• returns true or false if aProperty is a property of anObject

» `eval(someString)`

• evaluates a someString as if it was JavaScript. i.e. eval("var x = 2+3") returns 5

## ⚙ Array Methods (basic)

💡 **Note:** index numbers for arrays start at 0

» `arr.length()`

» `arr. push(val)`

• adds val to end of arr

» `arr. pop()`

• deletes last item in arr

» `arr. shift()`

• deletes first item in arr

» `arr.unshift(val)`

• adds val to front of arr

» `arr.reverse ()`

» `arr1.concat(arr2)`

• concatenates arr1 with arr2

» `arr.join(char)`

• returns string of elements of arr joined by char

» `arr.slice(index1, index2)`

• returns a new array from arr from index1 (inclusive) to index2 (exclusive)

» `arr.splice(index, num, itemA, itemB,..)`

• alters arr. starting at index and through index+num, overwrites/adds itemsA..

## 🎓 Definitions & Lingo

**Higher Order Functions**

functions that accept *other functions* as an argument

**Scope**

the set of variables, objects, and functions available within a certain block of code

**Callback**

(also *event handler*) a reference to executable code, or a piece of executable code, that is passed as an argument to other code.

**the % operator**

% returns the remainder of a division such that "3 % 2 = 1" as 2 goes into 3 once leaving 1. called the "remainder" or "modulo" operator.

**Composition**

the ability to assemble complex behaviour by aggregating simpler behavior. *chaining* methods via dot syntax is one example.

## 🎓 Definitions & Lingo (cont)

**Chaining**

also known as *cascading*, refers to repeatedly calling one method after another on an object, in one continuous line of code.

**Naming Collisions**

where two or more identifiers in a given namespace or a given scope cannot be unambiguously resolved

**DRY**

Don't Repeat Yourself

**ECMAScript**

(also *ECMA-262*) the specification from which the JavaScript implementation is derived. version 5.1 is the current release.

**Arity**

refers to the number of arguments an operator takes. ex: a binary function takes two arguments

**Currying**

refers to the process of transforming a function with multiple arity into the same function with less *arity*

**Recursion**

an approach in which a function calls itself

**Predicate**

a calculation or other operation that would evaluate either to "true" or "false."

**Asynchronous**

program flow that allows the code following an *asynchronous* statement to be executed immediately without waiting for it to complete first.

**Callback Hell**

code thickly nested with callbacks within callback within callbacks.

**Closure**

a function with access to the global scope, it's parent scope (if there is one), and it's own scope. a closure may retain those scopes even after it's parent function has *returned.*

**IIFE**

Immediately Invoked Function Expressions. *pronounced "iffy."* a function that is invoked immediately upon creation. employs a unique syntax.

## 🎓 Definitions & Lingo (cont)

**Method**

an object property has a function for its value.

## 🔒 Reserved Words

| abstract | arguments | boolean | break |
| --- | --- | --- | --- |
| byte | case | catch | char |
| class | const | continue | debugger |
| default | delete | do | double |
| else | enum | eval | export |
| extends | false | final | finally |
| float | for | function | goto |
| if | implements | import | in |
| instanceof | int | interface | let |
| long | native | new | null |
| package | private | protected | public |
| return | short | static | super |
| switch | synchronized | this | throw |
| throws | transient | true | try |
| typeof | var | void | volatile |
| while | with | yield | |

## 👥 Prototype-based Inheritance

*coming soon*