Introduction to

JavaScript

the programming language of the Web

Javascript built-in objects

Language Objects

Types

JavaScript types can be divided into two categories:

- *primitive types.*
  - number.
  - boolean.
  - string.
  - undefined.
- *object types*.
  - Language objects
  - Document objects
  - null
  - Browser objects
  - User-defined objects

Language Objects

- String
- Number
- Boolean
- Array
- Date
- Math

String

String Operations

```
var s = new String("hello, world“)        Start with some text.
s.charAt(0)               => "h": the first character.
s.charAt(s.length-1)      => "d": the last character.
s.substring(1,4)          => "ell": the 2nd, 3rd and 4th characters.
s.slice(1,4)              => "ell": same thing
s.slice(-3)               => "rld": last 3 characters
s.indexOf("l")            => 2: position of first letter l.
s.indexOf("M")            => returns -1 Not Found
s.lastIndexOf("l")        => 10: position of last letter l.
s.indexOf("l", 3)         => 3: position of first "l" at or after 3
s.split(", ")             => ["hello", "world"] split into substrings
s.replace("h", "H")       => "Hello, world": replaces all instances
s.toUpperCase()           => "HELLO, WORLD"
```

Write a JavaScript function that accepts a string as a parameter and counts the number of vowels within the string.

```
//Example : 'The quick brown fox'
//Output  : 5
function vowel_count(str1) {
    var vowel_list = 'aeiouAEIOU';
    var vcount = 0;
    for (var x = 0; x < str1.length ; x++) {
        if (vowel_list.indexOf(str1[x]) !== -1) {
            vcount += 1;
        }
    }
    return vcount;
}
console.log(vowel_count("The quick brown fox"));
```

Boolean

Boolean

The Boolean object is used to convert a non-Boolean value to a Boolean value (true or false).

- All the following lines of code create Boolean objects with an initial value of **false**:

```
var myBoolean=new Boolean();
var myBoolean=new Boolean(0);
var myBoolean=new Boolean(null);
var myBoolean=new Boolean("");
var myBoolean=new Boolean(false);
```

- All the following lines of code create Boolean objects with an initial value of **true**:

```
var myBoolean=new Boolean(true);
var myBoolean=new Boolean("true");
var myBoolean=new Boolean("false");
var myBoolean=new Boolean(5);
var myBoolean=new Boolean("Ali");
```

Number

Number

The **Number** JavaScript object is a wrapper object allowing you to work with numerical values. A Number object is created using the Number() constructor.

Methods

Number.isNaN() Determine whether the passed value is NaN.

Number.isFinite() Determine whether the passed value is a finite number.

Number.isInteger() Determine whether the passed value is an integer.

Number.isSafeInteger() Determine whether the passed value is a safe integer (number between $-(2^{53} - 1)$ and $2^{53} - 1$).

Number.parseFloat() The value is the same as parseFloat() of the global object.

Number.parseInt() The value is the same as parseInt() of the global object.

Convert numeric strings to numbers

```
Number("123")      123
Number("12.3")     12.3
Number("")         0
Number("0x11")     17
Number("foo")      NaN
Number("100a")     NaN
```

Explicit Conversions

```
Number("3")  => 3
String(false)  => "false" Or use false.toString()
Boolean("JS")=> true
```

```
var n = 17;
binary_string = n.toString(2);  Evaluates to "10001"
octal_string = "0" + n.toString(8);  Evaluates to "021"
hex_string = "0x" + n.toString(16);  Evaluates to "0x11"
```

Explicit Conversions

```
parseInt("3 dollars")              => 3
parseFloat(" 3.14 meters")         => 3.14
parseInt("-12.34")                 => -12
parseInt("0xFF")                   => 255
parseInt("0xff")                   => 255
parseInt("-0XFF")                  => -255
parseFloat(".1")                   => 0.1
parseInt("0.1")                    => 0
parseInt(".1")                     => NaN: integers can't start with "."
parseFloat("$72.47");              => NaN: numbers can't start with "$"
```

Type Conversion Table

Original Value	to Number	to String	to Boolean
false	0	"false"	false
true	1	"true"	true
0	0	"0"	false
1	1	"1"	true
"0"	0	"0"	true
"1"	1	"1"	true
NaN	NaN	"NaN"	false
Infinity	Infinity	"Infinity"	true
-Infinity	-Infinity	"-Infinity"	true

JS

Type Conversion Table

Original Value	to Number	to String	to Boolean
""	**0**	""	**false**
"20"	20	"20"	true
"twenty"	NaN	"twenty"	true
[ ]	**0**	""	true
[20]	**20**	"20"	true
[10,20]	NaN	"10,20"	true
["twenty"]	NaN	"twenty"	true
["ten","twenty"]	NaN	"ten,twenty"	true
function(){}	NaN	"function(){}"	true

Type Conversion Table

Original Value	to Number	to String	to Boolean
{ }	NaN	"[object Object]"	true
null	**0**	"null"	false
undefined	NaN	"undefined"	false

The + Operator

The binary + operator adds numeric operands or concatenates string operands

```
1 + 2  => 3
"hello" + " " + "there"  => "hello there"
"1" + "2"  => "12“
```

```
1 + 2  => 3: addition
"1" + "2"  => "12": concatenation
"1" + 2  => "12": concatenation after number-to-string
1 + {}  => "1[object Object]": concatenation after object-to-string
true + true  => 2: addition after boolean-to-number
2 + null  => 2: addition after null converts to 0
2 + undefined  => NaN: addition after undefined converts to NaN
```

Arrays

Arrays

- An *array* is an ordered collection of values.
- Each value is called an *element*, and each element has a numeric position in the array, known as its **index**.
- JavaScript arrays are **un-typed**: an array element may be of any type, and different elements of the same array may be of different types.
- Array elements may even be objects or other arrays, which allows you to create complex data structures, such as arrays of objects and arrays of arrays.
- JavaScript arrays are **zero-based** and use 32-bit indexes: the index of the first element is 0, and the highest possible index is 4,294,967,295.
- JavaScript arrays are **dynamic**: they grow or shrink as needed and there is no need to declare a fixed size for the array when you create it or to reallocate it when the size changes.
- Every JavaScript array has a **length** property. this property specifies the number of elements in the array.

Creating Arrays

Array literal syntax

```
var empty = [];  An array with no elements
var primes = [2, 3, 5, 7, 11];  An array with 5 numeric elements
var misc = [ 1.1, true, "a", ];  3 elements of various types + trailing comma
```

Array() constructor

```
var a = new Array();        Equal to []
var a = new Array(10);
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

Reading and Writing Array Elements

```
var a = ["world"];  Start with a one-element array
var value = a[0];  Read element 0
a[1] = 3.14;  Write element 1
i = 2;
a[i] = 3;  Write element 2
a[i + 1] = "hello";  Write element 3
a[a[i]] = a[0];  Read elements 0 and 2, write element 3
a[1000] = 0;  Assignment adds one element but sets length to 1001.
```

Array Length

```
[].length  => 0: the array has no elements
['a','b','c'].length  => 3: highest index is 2, length is 3

a = [1,2,3,4,5];  Start with a 5-element array.
a.length = 3;  a is now [1,2,3].
a.length = 0;  Delete all elements. a is [].
a.length = 5;  Length is 5, but no elements, like new Array(5)
```

Adding and Deleting Array Elements

```
a = []  Start with an empty array.
a[0] = "zero";  And add elements to it.
a[1] = "one";

a = [];  Start with an empty array
a.push("zero")  Add a value at the end. a = ["zero"]
a.push("one", "two")  Add two more values. a = ["zero", "one", "two"]

a = [1,2,3];
delete a[1];  a now has no element at index 1
1 in a  => false: no array index 1 is defined
a.length  => 3: delete does not affect array length
```

Iterating Arrays

```
var a = [1,2,3,,5,6];
a[10] = 10;

for(var i = 0; i < a.length; i++) {
    if (!a[i]) continue;  Skip null, undefined, and nonexistent elements
        console.log(a[i]);
}

for(i in a)
    console.log(a[i]);
```

Which array method to us?

To mutate original array

- Add to original:
  - `.push` *(end)*
  - `.unshift` *(start)*
- Remove from original:
  - `.pop` *(end)*
  - `.shift` *(start)*
  - `.splice` *(any)*
- Others:
  - `.reverse`
  - `.sort`
  - `.fill`

A new array

- Computed from original:
  - `.map` *(loop)*
- Filtered using condition:
  - `.filter`
- Portion of original:
  - `.slice`
- Adding original to other:
  - `.concat`
- Flattening the original:
  - `.flat`
  - `.flatMap`

An array index

- Based on value:
  - `.indexOf`
- Based on test condition:
  - `.findIndex`

An array element

- Based on test condition:
  - `.find`

Know if array includes

- Based on value:
  - `.includes`
- Based on test condition:
  - `.some`
  - `.every`

A new string

- Based on separator string:
  - `.join`

To transform to value

- Based on accumulator:
  - `.reduce`

(Boil down array to single value of any type: number, string, boolean, or even new array or object)

To just loop array

- Based on callback:
  - `.forEach`

(Does not create a new array, just loops over it)

Array.foreach

The forEach() method executes a provided function once per array element.

Syntax :

arr.forEach(callback

Parameters :

Callback : Function to execute for each element, taking three arguments:

1. currentValue : The current element being processed in the array.
2. Index : The index of the current element being processed in the array.
3. Array : The array filter was called upon.

Return value :

undefined.

forEach() executes the provided callback once for each element present in the array in ascending order. It is not invoked for index properties that have been deleted or are uninitialized (i.e. on sparse arrays).

Array.foreach

There is no way to stop or break a forEach() loop other than by throwing an exception. If you need such behavior, the forEach() method is the wrong tool, use a plain loop instead. If you are testing the array elements for a predicate and need a Boolean return value, you can use every() or some() instead.

Examples : Printing the contents of an array

The following code logs a line for each element in an array:

```
function logArrayElements(element, index, array) {
    console.log('a[' + index + '] = ' + element);
}
// Notice that index 2 is skipped since there is no item at that position in the array.
[2, 5, , 9].forEach(logArrayElements);
// a[0] = 2
// a[1] = 5
// a[3] = 9
```

Array Methods

Array.join() method converts all the elements of an array to strings and concatenates them, returning the resulting string. You can specify an optional string that separates the elements in the resulting string. If no separator string is specified, a comma is used.

```
var a = [1, 2, 3];  Create a new array with these three elements
a.join();  => "1,2,3"
a.join(" ");  => "1 2 3"
a.join("");  => "123"
var b = new Array(10);  An array of length 10 with no elements
b.join('-')  => '---------': a string of 9 hyphens
```

Array Methods

Array.reverse() method reverses the order of the elements of an array and returns the reversed array. it doesn't create a new array.

```
var a = [1,2,3];

a.reverse().join()  => "3,2,1" and a is now [3,2,1]
```

Array.filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

Syntax :

```
var new_array = arr.filter(callback)
```

Parameters :

Callback : Function is a predicate, to test each element of the array. Return true to keep the element, false otherwise, taking three arguments:

1. Element : The current element being processed in the array.
2. Index : The index of the current element being processed in the array.
3. Array : The array filter was called upon.

Return value :

A new array with the elements that pass the test.

Array.filter

filter() calls a provided callback function once for each element in an array, and constructs a new array of all the values for which callback returns a value that coerces to true. callback is invoked only for indexes of the array which have assigned values; it is not invoked for indexes which have been deleted or which have never been assigned values. Array elements which do not pass the callback test are simply skipped, and are not included in the new array.

Examples : Filtering out all small values

The following example uses filter() to create a filtered array that has all elements with values less than 10 removed.

```
function isBigEnough(value) {
    return value >= 10;
}
var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
// filtered is [12, 130, 44]
```

Array Methods

Array.sort() sorts the elements of an array in place and returns the sorted array. When sort() is called with no arguments, it sorts the array elements in alphabetical order.

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", ");  s == "apple, banana, cherry“


var a = [33, 4, 1111, 222];
a.sort();                  Alphabetical order: 1111, 222, 33, 4
a.sort(function(a,b) {     Numerical order: 4, 33, 222, 1111
    return a-b;            Returns <0, 0, or >0, depending on order
});
a.sort(function(a,b) {return b-a});  Reverse numerical order
```

Array Methods

```
a = ['ant', 'Bug', 'cat', 'Dog']
a.sort();  case-sensitive sort: ['Bug','Dog','ant',cat']
a.sort(function(s,t) {  Case-insensitive sort
    var a = s.toLowerCase();
    var b = t.toLowerCase();
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
});  => ['ant','Bug','cat','Dog']
```

Write a JavaScript function to get the first elements of an array.

Passing a parameter 'n' will return the first 'n' elements of the array

```
first = function (array, n) {
    if (n < 0)
        return [];
    return array.slice(0, n);
};
```

Write a JavaScript function to get the last elements of an array.

Passing a parameter 'n' will return the last 'n' elements of the array

```
last = function (array, n) {
    if (n > 0)
        return array.slice(array.length - n);
};
```

```
function equalArrays(a,b)
{
    if (a.length != b.length)
        return false;              Different-size arrays not equal
    for(var i = 0; i < a.length; i++)
    {                              Loop through all elements
        if (a[i] !== b[i])
        return false;              If any differ, arrays not equal
    }
    return true;                   Otherwise they are equal
}
```

Multidimensional Arrays

```
 Create a multidimensional array
var table = new Array(10);  10 rows of the table
for(var i = 0; i < table.length; i++)
    table[i] = new Array(10);  Each row has 10 columns

 Initialize the array
for(var row = 0; row < table.length; row++) {
    for(col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}
 Use the multidimensional array to compute 5*7
var product = table[5][7];  35
```

Function Blocks: Rest and Spread

JS

Rest Parameters

A **rest** parameter is indicated by **three dots (...)** preceding a named parameter.

That named parameter becomes an **Array** containing the rest of the

parameters passed to the function, which is where the name rest parameters originates.

```
function addOrder(orderId,...products)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
addOrder(2,"item1","item2");
```

Rest Parameter Restrictions

The first restriction is that there **can be only one rest parameter**, and the rest parameter **must be last parameter**.

```
function addOrder(orderId,...products,category)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
```

Or

```
function addOrder(orderId,...products,...category)
{
    console.log(products.constructor.name); //Array
    //do smothing
}
```

Syntax Error → Rest parameter must be last formal parameter.

The Spread Operator

The spread operator allows you to specify an array that should be split and passed in as separate arguments to a function.

Consider the built-in Math.max() method, which accepts any number of arguments and returns the one with the highest value.

```
var numbers = [3, 1, 7, 4, 9];
console.log(Math.max(3,1,7,4,9)); //9
//or
console.log(Math.max.apply(null,numbers)); //9
```

you can pass the array to Math.max() directly and prefix it with the same ... pattern you use with rest parameters. The JavaScript engine then splits the array into individual arguments and passes them in.

```
console.log(Math.max(…numbers)); //9
```

We can use spread inside another array as follow

```
var AllNumbers = [33, 55, 11, ...numbers, 90]; // [33,55,11,3,1,7,4,9,90]
```

Math

Math

JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the Math object:

```
Math.pow(2,53)          => 9007199254740992: 2 to the power 53
Math.round(.6)          => 1.0: round to the nearest integer
Math.ceil(.6)           => 1.0: round up to an integer
Math.floor(.6)          => 0.0: round down to an integer
Math.abs(-5)            => 5: absolute value
Math.max(x,y,z)         Return the largest argument
Math.min(x,y,z)         Return the smallest argument
Math.random()           Pseudo-random number x where 0 <= x < 1.0
Math.PI                 π: circumference of a circle / diameter
Math.E                  e: The base of the natural logarithm
```

Math

```
Math.sqrt(3)              The square root of 3
Math.pow(3, 1/3)          The cube root of 3
Math.sin(0)               Trigonometry: also Math.cos, Math.atan, etc.
Math.log(10)              Natural logarithm of 10
Math.log(100)/Math.LN10   Base 10 logarithm of 100
Math.log(512)/Math.LN2    Base 2 logarithm of 512
Math.exp(3)               Math.E cubed
```

Date

Date

Creates a JavaScript **Date** instance that represents a single moment in time.

Date objects are based on a time value that is the number of milliseconds since **1 January, 1970 UTC.**

Syntax:

```
new Date();
new Date(value);
new Date(dateString);
new Date(year, month[, day[, hour[, minutes[, seconds[, milliseconds]]]]]);
```

```
Date.getDate()
//Returns the day of the month (1-31) for the specified date according to local time.
Date.getDay()
//Returns the day of the week (0-6) for the specified date according to local time.
Date.getFullYear()
//Returns the year (4 digits years) of the specified date according to local time.
Date.getHours()
//Returns the hour (0-23) in the specified date according to local time.
Date.getMilliseconds()
//Returns the milliseconds (0-999) in the specified date according to local time.
Date.getMinutes()
//Returns the minutes (0-59) in the specified date according to local time.
Date.getMonth()
//Returns the month (0-11) in the specified date according to local time.
Date.getSeconds()
//Returns the seconds (0-59) in the specified date according to local time.
Date.getTime()
//Returns the numeric value of the specified date as the number of milliseconds since January 1,
1970, 00:00:00 UTC (negative for prior times).
```

```
Date.setDate()
//Sets the day of the month for a specified date according to local time.
Date.setFullYear()
//Sets the full year (e.g. 4 digits for 4-digit years) for a specified date according to local
time.
Date.setHours()
//Sets the hours for a specified date according to local time.
Date.setMilliseconds()
//Sets the milliseconds for a specified date according to local time.
Date.setMinutes()
//Sets the minutes for a specified date according to local time.
Date.setMonth()
//Sets the month for a specified date according to local time.
Date.setSeconds()
//Sets the seconds for a specified date according to local time.
Date.setTime()
//Sets the Date object to the time represented by a number of milliseconds since January 1, 1970,
00:00:00 UTC, allowing for negative numbers for times prior.
```

```
Date.toDateString()
//Returns the "date" portion of the Date as a human-readable string.
Date.toISOString()
//Converts a date to a string following the ISO 8601 Extended Format.
Date.toJSON()
//Returns a string representing the Date using toISOString().
Date.toGMTString()
//Returns a string representing the Date based on the GMT (UT) time zone. Use toUTCString() instead.
Date.toLocaleString()
//Returns a string with a locality sensitive representation of this date. Overrides the
Object.toLocaleString() method.
Date.toString()
//Returns a string representing the specified Date object. Overrides the Object.toString() method.
Date.toTimeString()
//Returns the "time" portion of the Date as a human-readable string.
Date.toUTCString()
//Converts a date to a string using the UTC timezone.
```