Eman Fathi

Introduction to

JavaScript

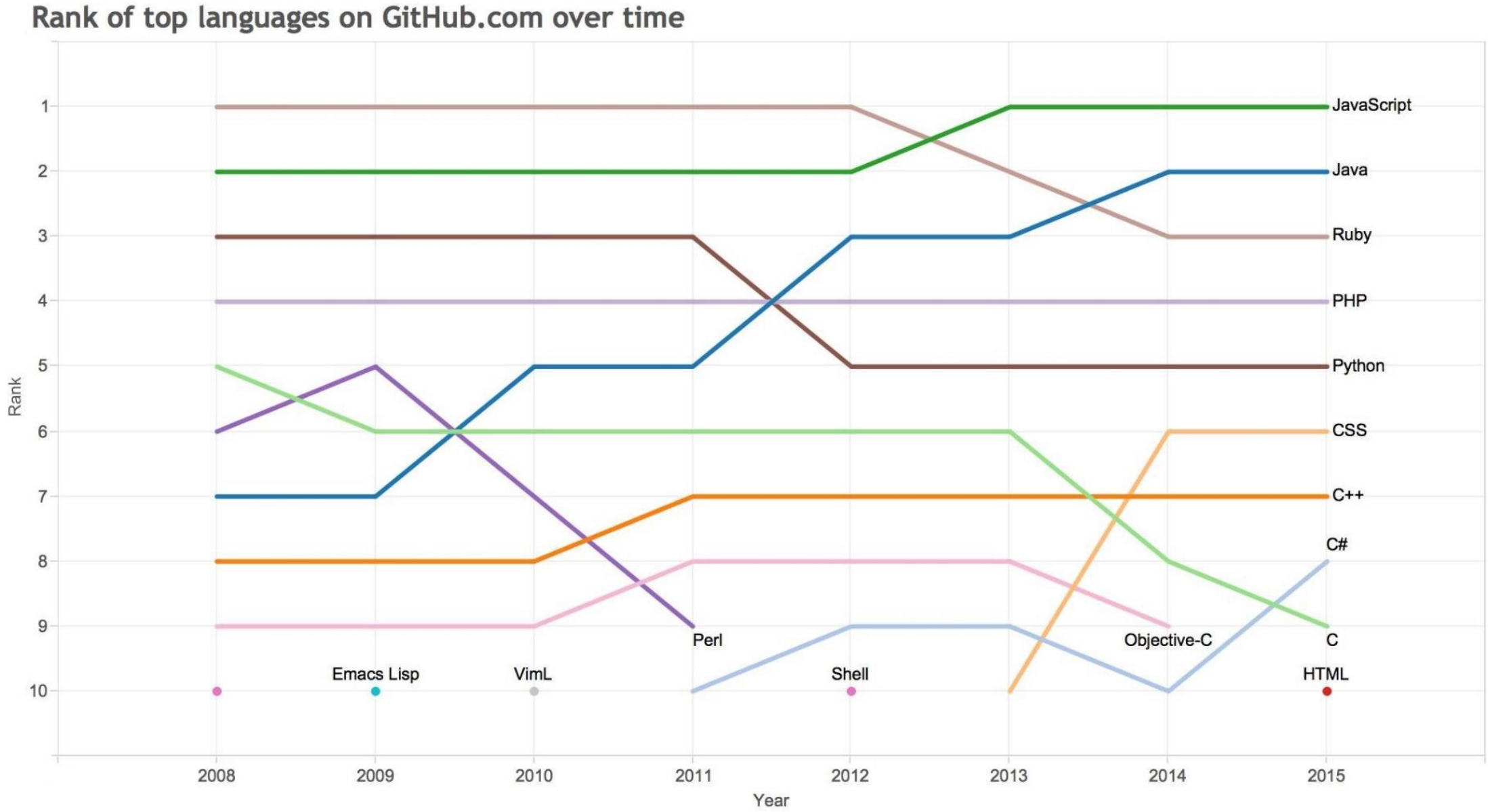
the programming language of the Web

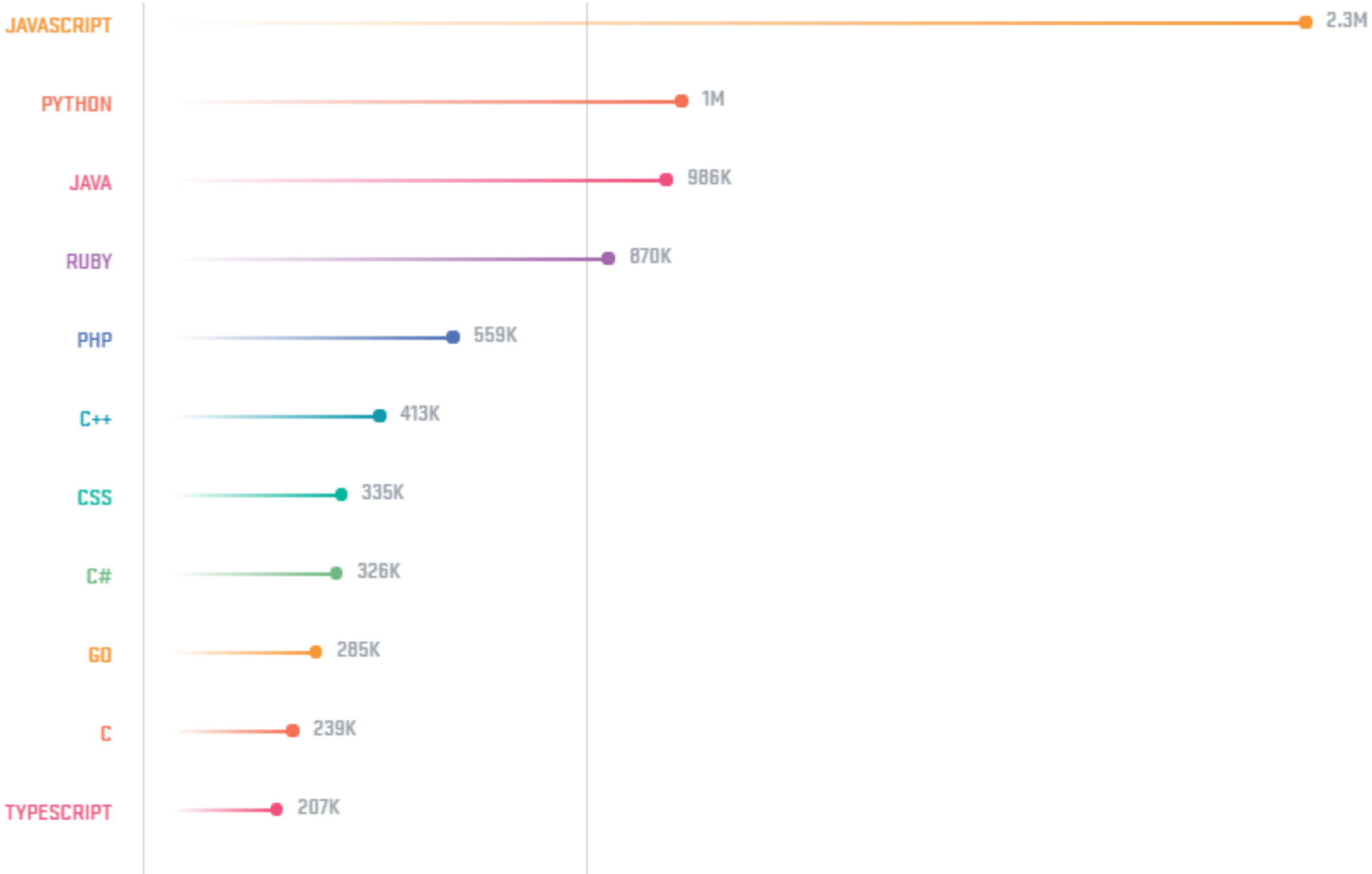
Agenda

- **Day 1** : Core JS
- **Day 2** : Browser Object Model
- **Day 3** : Document Object Model
- **Day 4** : Events
- **Day 5** : DHTML

Course Assessment

- Labs : 40%
- Project : 60%


Rank of top languages on GitHub.com over time
Rank
1
2
3
4
5
6
7
8
9
10
JavaScript
Java
Ruby
PHP
Python
CSS
C++
C#
C
HTML
Perl
Objective-C
Emacs Lisp
VimL
Shell
2008
2009
2010
2011
2012
2013
2014
2015
Year
Source: GitHub.com


JAVASCRIPT
2.3M
PYTHON
1M
JAVA
986K
RUBY
870K
PHP
559K
C++
413K
CSS
335K
C#
326K
GO
285K
C
239K
TYPESCRIPT
207K

What you should already know

- A general understanding of the **Internet** and the **World Wide Web** (WWW).
- Good working knowledge of **HyperText Markup Language** (HTML).
- Some **programming experience**.

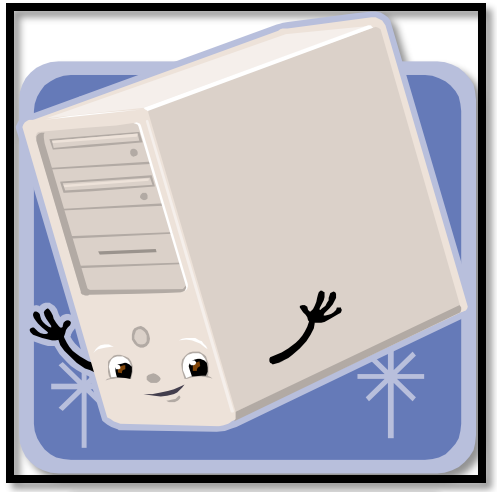
We need to turn them into developers who think of applications like this.

How Does the Web Work?

Client

http

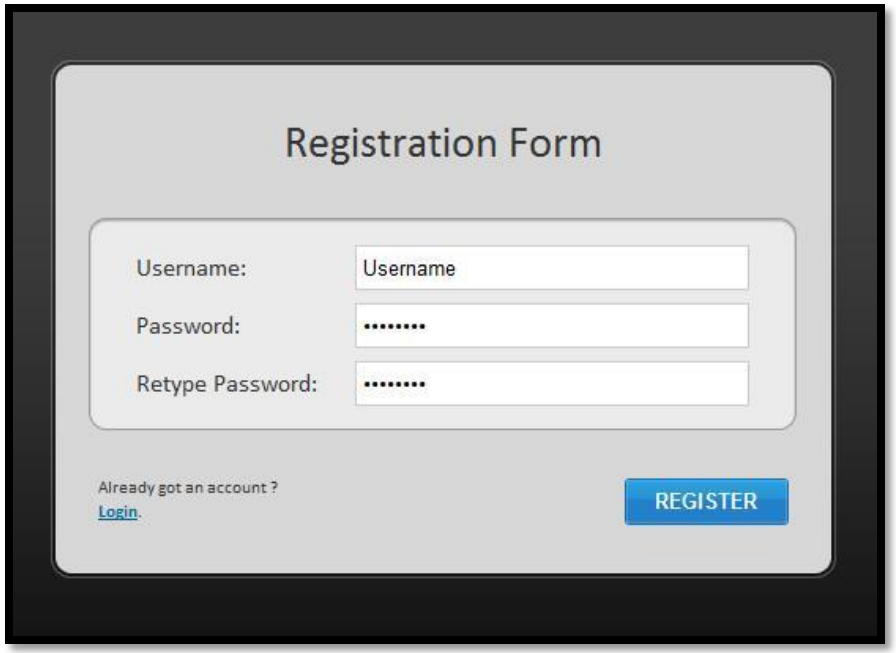
html



Server

StaticWebPage

Traffic



Send Data

Send Page

If(userName==“”)

If(Pass==“”)

If(ConfPass==“”)

If(Pass==CnfPass)

JS


Browser
Wars!

JavaScript history

Brendan Eich convinced his boss at **Netscape** that the Navigator browser should have its own **scripting language**, and that only a new language would do, a new language designed and implemented in big hurry, and that no existing language should be considered for that role.

Brendan Eich

Introduction to JavaScript

- JavaScript is *the* language of the Web. It was introduced in 1995 by **Netscape**, the company that created the first browser by the same name. Other browsers followed, and JavaScript quickly gained acceptance as the client-side scripting language on the internet. Today it is used by millions of websites to add **functionality** and **responsiveness**, **validate forms**, **communicate with the server**, and much more.
- Originally, JavaScript was named **LiveScript** but was renamed **JavaScript** as a marketing strategy to benefit from the exploding popularity of the **Java** programming language at that time. As it turned out, Java evolved into a server-side language and did not succeed on the browser, whereas JavaScript did. The change of name is unfortunate because it has caused a lot of confusion.

Introduction to JavaScript

- Soon after its initial release the JavaScript language was submitted to **ECMA International** -- an international non-profit standards organization -- for consideration as an industry standard. It was accepted and today we have a standardized version which is called **ECMAScript**. There are several implementations of the ECMAScript standard, including **JavaScript**, **Jscript** (Microsoft), and **ActionScript** (Adobe). The ECMAScript language is undergoing continuous improvements. The next standards release is ECMAScript **6th** edition and is code named "**Harmony**".
- Initially many developers felt it was an inferior language because of its perceived **simplicity**. Also, users would frequently disable JavaScript in their browsers because of security concerns. However, over the last few years, starting with the introduction of **AJAX** and the related Web 2.0 transition, it became clear that JavaScript allows developers to **build powerful and highly responsive web applications**. Today JavaScript is considered *the* Web language and an essential tool for building modern and responsive web apps.

Historically JavaScript was not really taken very seriously.

JavaScript

- JavaScript is a **cross-platform**, **object-oriented scripting language**.
- It is a **small** and **lightweight** language. **Inside a host environment** (for example, a **web browser**)
- JavaScript can be connected to the objects of its environment to provide programmatic control over them.

JavaScript

- JavaScript contains a standard library of objects, such as **Array**, **Date**, and **Math**, and a core set of language elements such as **operators**, **control structures**, and **statements**. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- Client-side JavaScript extends the core language by **supplying objects** to control a **browser** and its **Document Object Model (DOM)**. For example, client-side extensions allow an application to place elements on an HTML form and **respond to user events** such as mouse clicks, form input, and page navigation.

Embedding JavaScript in HTML

Client-side JavaScript code is embedded within HTML documents in four ways:

- Inline, between a pair of <script> and </script> tags
- From an external file specified by the src attribute of a <script> tag
- In an HTML event handler attribute, such as onclick or onmouseover

The <script> Element

JavaScript code can appear inline within an HTML file between **<script>** and **</script>** tags.

```
<!DOCTYPE html> <!-- This is an HTML5 file -->
<html>  <!-- The root element -->
<head>  <!-- Title, scripts & styles go here -->
<script>
        A script of JavaScript code
</script>
</head>
<body>  <!-- The body is the displayed parts of the doc. -->
<script>
        A script of JavaScript code
</script>
</body>
</html>
```

Scripts in External Files

The <script> tag supports a **src** attribute that specifies the URL of a file containing JavaScript code.

```
<!DOCTYPE html> <!-- This is an HTML5 file -->
<html>  <!-- The root element -->
<head>  <!-- Title, scripts & styles go here -->
    <title>Home Page</title>
    <script src="../../scripts/util.js"></script>
</head>
<body>  <!-- The body is the displayed parts of the doc. -->
    <h1>Welcome to JS.</h1>
    <script>
        A script of js code
    </script>
</body>
</html>
```

Event Handlers in HTML

JavaScript functions that are registered with the web browser and then invoked by the web browser in response to events (such as user input).

Event handler properties like onclick mirror HTML attributes with the same names, and it is also possible to define event handlers by placing JavaScript code in HTML attributes.

```
<input type="button" id="btn" value="click here"
       onclick="alert('click again'); this.disabled = true;">
```

Language Basics

- JavaScript is **case-sensitive**.
- In JavaScript, instructions are called statements and are separated by a semicolon (;).
- Spaces, tabs and newline characters are called whitespace.
- The source text of JavaScript scripts gets scanned from left to right and is converted into a sequence of input elements which are **tokens**, **control characters**, **line terminators**, **comments** or **whitespace**.
- ECMAScript also defines certain keywords and literals and has rules for automatic insertion of semicolons (ASI) to end statements. However, it is **recommended to always add semicolons to end your statements**; it will avoid side effects.

Case Sensitivity

JavaScript is a case-sensitive language.

This means that language keywords, variables, function names, and other *identifiers* must always be typed with a consistent capitalization of letters.

- Example while keyword, must be typed "while," not "While" or "WHILE."
- **online**, **Online**, **OnLine**, and **ONLINE** are four distinct variable names.

Comments

JavaScript supports two styles of comments.

- Any text between a and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is also treated as a comment; these comments may span multiple lines.

```
 This is a single-line comment.
/* This is also a comment */      and here is another comment.
/*
* This is yet another comment.
* It has multiple lines.
*/
```

LITERALS

A *literal* is a data value that appears directly in a program. The following are all literals:

```
12              The number twelve
1.2             The number one point two
"hello world"   A string of text
'Hi'            Another string
true            A Boolean value
false           The other Boolean value
/javascript/gi  A "regular expression" literal
null            Absence of an object
```

Reserved Words

JavaScript reserves a number of identifiers as the keywords of the language itself. You cannot use these words as identifiers in your programs:

break delete function return typeof case do if switch var catch else in this void continue

false instanceof throw while debugger finally new true with default for null try

Identifiers and Reserved Words

An identifier is simply a name. In JavaScript, identifiers are used to **name variables** and **functions** in JavaScript code.

A JavaScript identifier must begin with a **letter**, an **underscore** (_), or a **dollar sign** ($). Subsequent characters can be letters, digits, underscores, or dollar signs.

(Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.)

These are all legal identifiers:

- `i`
- `my_variable_name`
- `v13`
- `_dummy`
- `$str`

Core JavaScript

- Types, Values, and Variables
- Expressions and Operators
- Statements
- Arrays
- Objects
- Functions
- Classes and Modules
- Pattern Matching with **Regular Expressions (Report)**

Types, Values, and Variables

Types

JavaScript types can be divided into two categories:

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

Primitive

A primitive (primitive value, primitive data type) is data that is not an object and has no methods. In JavaScript, there are 5 primitive data types: **string**, **number**, **boolean**, **null**, **undefined.**

Most of the time, a primitive value is represented directly at the lowest level of the language implementation.

- All primitives are **immutable** (cannot be changed).

Primitive wrapper objects

Except for null and undefined, all primitive values have object equivalents that wrap around the primitive values:

- **String** for the string primitive.
- **Number** for the number primitive.
- **Boolean** for the Boolean primitive.
- The wrapper's **valueOf**() method returns the primitive value.

Variable Declaration

Before you use a variable in a JavaScript program, you should *declare* it. Variables are declared with the **var** keyword, like this:

```
var i;

var sum;
```

You can also declare multiple variables with the same var keyword:

```
var i, sum;
```

And you can combine variable declaration with variable initialization:

```
var message = "hello";

var i = 0, j = 0, k = 0;
```

If you don't specify an initial value for a variable with the var statement, the variable is declared, but its value is undefined until your code stores a value into it.

number

JavaScript does not make a distinction between integer values and floating-point values.
All numbers in JavaScript are represented as floating-point values.
Number is a numeric data type in the [double-precision 64-bit floating point format](#).

```
Integer
0
3500
10000000
Floating-Point
3.14
2345.789
.33333333333333333
6.02e23           6.02 × 1023
1.4738223E-32     1.4738223 × 10-32
```

string

Set of characters enclosed within a matched pair of single or double quotes (' or ").

```
""  The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\n has two lines"
"π is the ratio of a circle's circumference to its diameter"
```

string

String literals must be written on a single line. In ECMAScript 5, however, you can break a string literal across multiple lines by ending each line except the last with a backslash (\).

```
"two\n lines"     A string representing 2 lines written on one line


 A one-line string written on 3 lines. ECMAScript 5 only.
"one\
long\
line";
```

Long literal strings

Sometimes, your code will include strings which are very long.

- You can use the + operator to append multiple strings together, like this:

```
var longString = "This is a very long string which needs "
        + "to wrap across multiple lines because "
        + "otherwise my code is unreadable.“;
```

- Or you can use the backslash character ("\") at the end of each line to indicate that the string will continue on the next line.

```
var longString = "This is a very long string which needs \
        to wrap across multiple lines because \
        otherwise my code is unreadable.";
```

Escape Sequences in String

The **backslash character** (\) has a special purpose in JavaScript strings.

Combined with the character that follows it, it represents a character that is not otherwise representable within the string. For example, \n is an *escape sequence* that represents a newline character.

the backslash allows you to escape from the usual interpretation of the single-quote character. Instead of using it to mark the end of the string, you use it as an apostrophe:

```
'You\'re right, it can\'t be a quote'
```

Escape Sequences in String

\0	The NUL character (\u0000)
\b	Backspace (\u0008)
\t	Horizontal tab (\u0009)
\n	Newline (\u000A)
\v	Vertical tab (\u000B)
\r	Carriage return (\u000D)
\"	Double quote (\u0022)
\'	Apostrophe or single quote (\u0027)
\\	Backslash (\u005C)

boolean

- Boolean is a logical data type that can have only the values **true** or **false**.
- JavaScript Booleans can have one of two values: **true** or **false**.

```
var completed = false;
var isMember = true;
if(isMember)
    discount = 15;
```

null

- The value null represents the intentional absence of any object value.

```
var element = null;
element = document.getElementById("i1");
```

undefined

- A primitive value automatically assigned to variables that have just been declared or to formal arguments for which there are no actual arguments.

```
var x;
console.log(x);


var temp;
typeof temp; //"undefined"
```

Difference between null and undefined

```
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
```

Javascript built-in objects

Language 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.

Syntax:

```
new Number(value);
```

Properties

Number.EPSILON The smallest interval between two representable numbers.

Number.MAX_VALUE The largest positive representable number.

Number.MIN_VALUE The smallest positive representable number - that is, the positive number closest to zero (without actually being zero).

Number.NaN Special "not a number" value.

Number.NEGATIVE_INFINITY Special value representing negative infinity; returned on overflow.

Number.POSITIVE_INFINITY Special value representing infinity; returned on overflow.

Number

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.

Number

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

Convert numeric strings to numbers

```
Number("123")      123
Number("12.3")     12.3
Number("")         0
Number("0x11")     17
Number("0b11")     3
Number("0o11")     9
Number("foo")      NaN
Number("100a")     NaN
```

Explicit Conversions

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

```
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

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
```

Operators

Arithmetic Operators

Arithmetic operators are used to perform arithmetic between variables and/or values.
Given that y = 5, the table below explains the arithmetic operators:

Operator	Description	Example	Result in y	Result in x
+	Addition	x = y + 2	y = 5	x = 7
-	Subtraction	x = y - 2	y = 5	x = 3
*	Multiplication	x = y * 2	y = 5	x = 10
/	Division	x = y / 2	y = 5	x = 2.5
%	Modulus (division remainder)	x = y % 2	y = 5	x = 1
++	Increment	x = ++y	y = 6	x = 6
		x = y++	y = 6	x = 5
--	Decrement	x = --y	y = 4	x = 4
		x = y--	y = 4	x = 5

Assignment operators

Operator	Example	Equivalent
+=	a += b	a = a + b
-=	a -= b	a = a – b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a & b	a = a & b
|=	a |= b	a = a | b
^=	a ^ b	a = a ^ b

JS

Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.
Given that x = 5, the table below explains the comparison operators:

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
===	equal value and equal type	x === "5"	false
		x === 5	true
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== "5"	true
		x !== 5	false
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	*true*

Logical Operators

Logical operators are used to determine the logic between variables or values.
Given that x = 6 and y = 3, the table below explains the logical operators:

Operator	Description	Example
&&	and	(x < 10 && y > 1) is true
||	or	(x === 5 || y === 5) is false
!	not	!(x === y) is true

Bitwise Operators

Bit operators work on 32 bits numbers. Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description	Example	Same as	Result	Decimal
&	AND	x = 5 & 1	0101 & 0001	0001	1
|	OR	x = 5 | 1	0101 | 0001	0101	5
~	NOT	x = ~ 5	~0101	1010	10
^	XOR	x = 5 ^ 1	0101 ^ 0001	0100	4
<<	Left shift	x = 5 << 1	0101 << 1	1010	10
>>	Right shift	x = 5 >> 1	0101 >> 1	0010	2

Evaluation Expressions

JavaScript has the ability to interpret strings of JavaScript source code, evaluating them to produce a value. JavaScript does this with the global function **eval**():

```
eval("3+2")  => 5
eval("var x = 100; alert('Hello');");
```

If you construct an arithmetic expression as a string, you can use eval() to evaluate it at a later time.

eval() used to be used to parse JSON strings, but that use has been superseded by the faster and more-secure **JSON.parseJSON**.

The Conditional Operator (?:)

The conditional operator is the only ternary operator (three operands) in JavaScript and is sometimes actually called the ternary operator.

Syntax: *condition ? expr1 : expr2*

```
var fee = (isMember ? "$2.00" : "$10.00")
==
if(isMember)
    fee = "$2.00";
else
    fee = "$10.00";
```

The typeof Operator

X	typeof X
undefined	"undefined"
null	"object"
true or false	"boolean"
any number or NaN	"number"
any string	"string"
any function	"function"
any nonfunction native object	"object"

Conditionals

if

The fundamental control statement that allows JavaScript to make **decisions**, or, more precisely, to **execute statements conditionally**. This statement has two forms.

if (*expression*)
** *statement***

```
if (username == null)    If username is null or undefined,
    username = "guest";  define it


if (!address) {
    address = "";
    message = "Please specify a mailing address.";
}
```

if

The second form of the if statement introduces an else clause that is executed when *expression* is false.

if (*expression*)
**   *statement1***
else
**   *statement2***

```
if (hour < 18) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

else if

```
if (n == 1) {
     Execute code block #1
}
else if (n == 2) {
     Execute code block #2
}
else if (n == 3) {
     Execute code block #3
}
else {
     If all else fails, execute block #4
}
```

switch

An if statement causes a branch in the flow of a program's execution, and you can use the else if idiom to perform a **multiway branch**. The switch statement handles exactly this situation.

```
switch(n) {
    case 1:  Start here if n == 1
         Execute code block #1.
        break; Stop here
    case 2:  Start here if n == 2
         Execute code block #2.
        break;  Stop here
    default:  If all else fails...
         Execute code block #4.
        break;  stop here
}
```

switch

Use the weekday number to get weekday name:

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    case 2:
        day = "Tuesday";
        break;
    case 3:
        day = "Wednesday";
        break;
    case 4:
        day = "Thursday";
        break;
    case 5:
        day = "Friday";
        break;
    case 6:
        day = "Saturday";
}
```

Loops

while

The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

```
var count = 0;
while (count < 10) {
    console.log(count);
    count++;
}
```

do/while

The **do...while statement** creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

```
var i = 0;
do {
    i += 1;
    console.log(i);
} while (i < 5);
```

for

The for statement provides a looping construct that is often more convenient than the while statement. The for statement simplifies loops that follow a common pattern. Most loops have a **counter variable** of some kind. This variable is **initialized** before the loop starts and is **tested before each iteration** of the loop. Finally, the counter variable is **incremented** or otherwise updated at the end of the loop body.

```
for(var count = 0; count < 10; count++)
        console.log(count);


var i,j;
for(i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

for/in

A for...in loop only iterates over enumerable properties.

```
for(var i = 0; i < a.length; i++)          Assign array indexes to variable i
    console.log(a[i]);                     Print the value of each array element
==
for(i in a) console.log(i);
```

break

The break statement, used alone, causes the innermost enclosing loop or switch statement to exit immediately.

```
for(var i = 0; i < a.length; i++) {
    if (a[i] == target) break;
}
```

continue

The continue statement is similar to the break statement. Instead of exiting a loop, however, continue restarts a loop at the next iteration.

```
for(i = 0; i < data.length; i++) {
    if (!data[i]) continue;       Can't proceed with undefined data
    total += data[i];
}
```

Functions

A function is a block of JavaScript code that is defined once but may be executed, or invoked, any number of times.

Defining Functions

A function declaration statement has the following syntax:

```
//function definition statement
function [Name]([param [, param  [..., param ]]]) {
        statements
}
```

- **Name** The function name. Can be omitted, in which case the function becomes known as an anonymous function.
- **Param** The name of an argument to be passed to the function. A function can have up to 255 arguments.
- **Statements** The statements comprising the body of the function.

return

A return statement within a function specifies the value of invocations of that function.

```
function square(x) { return x * x; }      A function that has a return statement
square(2);                                This invocation evaluates to 4
```

Defining Functions

```
function definition statement
function Square(x) { return x * x; }
```

OR

```
function declaration expression
Note that we assign it to a variable
var square = function(x) { return x * x; }
```

Variable Scope

```
var scope = "global";           Declare a global variable
function checkscope() {
    var scope = "local";        Declare a local variable with the same name
    return scope;               Return the local value, not the global one
}
checkscope()                    => "local"
```

Variable Scope

Although you can get away with not using the var statement when you write code in the global scope, you must always use var to declare local variables.

```
scope = "global";          Declare a global variable, even without var.
function checkscope2() {
    scope = "local";       Oops! We just changed the global variable.
    myscope = "local";     This implicitly declares a new global variable.
    return [scope, myscope];  Return two values.
}
checkscope2()              => ["local", "local"]: has side effects!
scope                      => "local": global variable has changed.
myscope                    => "local": global namespace cluttered up.
```

Variable Hoisting

JavaScript code behaves as if all variable declarations in a function (but not any associated assignments) are “hoisted” to the top of the function.

```
var scope = "global";
function f() {
    console.log(scope);  Prints "undefined", not "global"
    var scope = "local";  Variable initialized here, but defined everywhere
    console.log(scope);  Prints "local"
}
```

Nested Functions

```
function hypotenuse(m, n) {          // outer function
    function square(num) {           // inner function
        return num * num;
    }
    return Math.sqrt(square(m) + square(n));
}
alert(hypotenuse(3,4));              // => 5
```

- The outer function hypotenuse contains an inner function square.
- The square function is visible only within the hypotenuse function body, that is, square has function scope only.
- The easiest way to look at square is as a private helper function to the outer function.

The Arguments Object

JavaScript functions have a built-in object called the arguments object.

The argument object contains an array of the arguments used when the function was called.

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

Default parameters

Default function parameters allow formal parameters to be initialized with default values if no value or undefined is passed.

```
function multiply(a, b) {
    var b = (typeof b !== 'undefined') ?  b : 1;
    return a*b;
}
 Default Parameters ES6
function multiply(a, b = 1) {
    return a*b;
}
multiply(5);  5
```

Objects

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]);
```

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.some

The some() method tests whether some element in the array passes the test implemented by the provided function.

Syntax :

```
arr.some(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 :

true if the callback function returns a truthy value for any array element; otherwise, false.

some() executes the callback function once for each element present in the array until it finds one where callback returns a truthy value (a value that becomes true when converted to a Boolean). If such an element is found, some() immediately returns true. Otherwise, some() returns false.

Array.some

Examples : Testing value of array elements

The following example tests whether any element in the array is bigger than 10.

```
function isBiggerThan10(element, index, array) {
    return element > 10;
}
[2, 5, 8, 1, 4].some(isBiggerThan10);  // false
[12, 5, 8, 1, 4].some(isBiggerThan10); // true
```

Array.every

The every() method tests whether all elements in the array pass the test implemented by the provided function.

Syntax :

arr.every(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 :

true if the callback function returns a truthy value for every element; otherwise, false.

The every method executes the provided callback function once for each element in the array until it finds one where callback returns a falsy value (a value that becomes false when converted to a Boolean). If such an element is found, the every method immediately returns false. Otherwise, if callback returned a true value for all elements, every will return true.

Array.some

Examples : Testing size of all array elements

The following example tests whether all elements in the array are bigger than 10.

```
function isBigEnough(element, index, array) {
    return element >= 10;
}
[12, 5, 8, 130, 44].every(isBigEnough);   // false
[12, 54, 18, 130, 44].every(isBigEnough); // true
```

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
```

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.
```

Write a JavaScript function to get the month name from a particular date

```
var month_name = function (dt) {
    mlist = ["January", "February", "March", "April", "May", "June", "July", "August",
"September", "October", "November", "December"];
    return mlist[dt.getMonth()];
}
```

Write a JavaScript function to test whether a date is a weekend

```
var is_weekend = function (date1) {
    var dt = new Date(date1);
    if (dt.getDay() == 6 || dt.getDay() == 0) {
        return "weekend";
    }
}
```

User-Defined Objects

OOP

Object-oriented programming (OOP)
- Creating reusable software objects

Object
- Programming code and data that can be treated as an individual unit or component

Data
- Information contained within variables

Object-oriented principles
- Encapsulation
- Inheritance
- Polymorphism

Instantiating an object

Creating an object from existing class

- Using **new** Operator

```
var today = new Date();
var objname = new Object();
```

To create an array you have two ways

```
var myarr = new Array[3];
//Or//both create array object
var myarr = [1,2,3];
```

From Arrays To Objects

In Arrays :

```
var myarr=[1,2,3];                    //using [] for array
```

In Objects :

```
var myobj={id:10};                    //using {} for object
```

This is an object that has a property called id with value = 10.

```
var a={};                             //a is an empty object
```

Encapsulation

Each object contain Properties & Methods to access these Properties.

```
var Car = { model: 'Toyota',//object has properties
    color: 'Red',
    move: function() {...}
    beep: function(r){ alert(r) }   //and methods
};
```

Accessing Object Members

There are two ways to access object members

- Using [ ] Square brackets.

```
Car['model'];              //get value from property
Car['color'] = "blue";     //set value to property
Car['move']();             //call a function
```

- Using **dot** operator.

```
Car.color="black";         //like C++
Car.move();
```

Altering Object Members

Because JavaScript code is parsed not compiled you can add or remove properties of the object.

- Add property to the object.

```
Car.motor="1300 cc";      //adding motor to Car
```

- Remove property from the object.

```
delete Car.model;         //deleting model attribute
Car.model;                //"undefined"
```

typeof Operator

Now we have the object Car but from which class we instantiated this object.

```
typeof Car;          //Object
```

Each object has a property called **Constructor**

```
Car.constructor;    // Object()
```

The object created using {} it's constructor is **Object()**

Classes in JavaScript

- JavaScript doesn't support the notion of *classes as typical OOP languages* do.

- In JavaScript, you **create *functions that can behave* *just* like classes** called **Constructor Function.**

- For example, you can call a function, or you can create an instance of that function supplying those parameters.

Constructor Function

```
function Human()
{
    this.name;                    // this refers to the caller object
    this.age;
    this.sayName=function()      // sayName function in human
    {alert(this.name);}
}
```

To take an object from Human

```
var obj = new Human();
obj.name="Mohamed";
obj.age=22;
obj.sayName();                    // alert Mohamed
```

Global Object

If we called Human function without new operator the **this** refers to the global object **Window**

When you say this.age you add age property to the window object.

```
Human();
Window.age=30;   //window has no age property
```

C++ & JavaScript Classes

```
Class Table
{
    public:
    int rows,cols;
    Table(int rows,int cols)
    {
        this.rows = rows;
        this.cols = cols;
    }
    int getCellCount()
    {
        return rows * cols;
    }
};
```

```
function Table(rows,cols)
{
    //constructor
    this.rows=rows;
    this.cols=cols;
    //method
    this.getCellCount=function()
    {
        return this.rows * this.cols;
    }
}
```

Private Members

```
function Table(rows,cols)
{
    //constructor
    var _rows = rows;
    var _cols = cols;
    //method
    this.getCellCount=function()
    {
        return _rows * _cols;
    }
}
```

```
//Now if you tries to access
var myTable=newTable(3,2);

myTable.getCellCount();
//it works and returns 6

mytable._rows;
//undefined
```

Object Literals

```
var empty = {};  An object with no properties
var point = { x:0, y:0 };  Two properties
var point2 = { x:point.x, y:point.y+1 };  More complex values
var book = {
    "main title": "JavaScript",  Property names include spaces,
    'sub-title': "The Definitive Guide",  and hyphens, so use string literals
    "for": "all audiences",  for is a reserved word, so quote
    author: {  The value of this property is
        firstname: "David",  itself an object. Note that
        surname: "Flanagan"  these property names are unquoted.
    }
};
```

Thank You