# What’s JavaScript?

JavaScript (JS) is a programming language primarily used by web browsers for dynamic scripting of web pages. It can also be used on the server side to perform any **sort** of action. Unlike most programming languages, the JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world. The most common host environment is the browser, but JavaScript interpreters can also be found in Adobe Acrobat, Photoshop, SVG images, Yahoo!'s Widget engine, as well as server side environments such as node.js. However the list of the areas where JavaScript is used just begins here. It also includes NoSQL databases, like the open source Apache CouchDB, embedded computers, or complete desktop environments, like GNOME (one of the most popular GUIs for GNU/Linux operating systems).

## Creator

JavaScript was created in 1995 by Brendan Eich, an engineer at Netscape, and first released with Netscape 2 early in 1996. And has several names like: Mocha, LiveScript, JScript, ECMAScript and JavaScript.

# Main Characteristics

## Is interpreted

This mean that Javascript code is never compiled to binary code, but read, understood and executed by a second program called “the interpreter”. In the case of web browsers, at least for the most popular ones, has a Javascript interpreter is embedded.

## Is object oriented

This is something people said very often about javascript, but it is not entirely true. Certainly, built-in objects are supported by javascript, even functions are treated as objects, but this language do not support natively object oriented features as classes, inheritance (at least not the class oriented one but the prototype oriented), function overloading, function overriding, polymorphism, among others. Some object oriented characteristics may be emulated using javascript, but this is something we are going to talk about later on.

## Is Case Sensitive

## This means that it makes a differentiation between identifiers taking into account the casing; for example the functions called **doWork** and **dowork** are different.

## Is Dynamic

Javascript is a dynamic language. That means that language has the ability to add code, decide data types, among other things, during execution time.

# Language Overview

As previously mentioned, JavaScript is an object oriented dynamic language; in this section will be described with certain detail level its main language components; that is, types and operators, standard built-in objects, programming structures and methods.

Javascript’s syntax style can be categorized as the C/C++ or Java type. This does not mean at all that Javascript is derived from those languages; just imply that many of their programming constructs looks very similar.

## Primitive data types

As previously mentioned, JavaScript is an object oriented language, and because of this all its primitive data types are objects, but classes are not supported; instead, the class functionality is accomplished by object prototypes. Another particularity is related to functions; in Javascript they are objects so they have the capacity to hold executable code and be passed around like any other object. The list below contains the primitive data types used in javascript:

* Number
* String
* Boolean
* Object
* Function
* Array
* Date
* RegExp
* null
* undefined

### Numbers

In JavaScript are "double-precision 64-bit format IEEE 754 values", according to the spec.

0.1 + 0.2 == 0.30000000000000004

Some build-in function:

* **Math** object
* **parseInt** and **parseFloat** (functions)
* unary **+** operator for convert
* **NaN** and **isNaN** (function)
* **Infinity**, minus **Infinity** and **isFinite**

### Strings

Strings in JavaScript are sequences of characters. More accurately, they are sequences of Unicode characters, with each character represented by a 16-bit number. This should be welcome news to anyone who has had to deal with internationalization.

Some build-in function:

* **chartAt** (function)
* **replace** (function)
* **toUpperCase** (function)

### Others types

JavaScript distinguishes between null, which is a value that indicates a deliberate non-value (and is only accessible through the null keyword), and undefined, which is a value of type 'undefined' that indicates an uninitialized value — that is, a value hasn't even been assigned yet.

(undefined is a constant)

JavaScript has a boolean type, with possible values true and false (both of which are keywords). Any value can be converted to a boolean according to the following rules:

* false, 0, the empty string (""), NaN, null, and undefined all become false.
* all other values become true.

(Use Boolean function for checking)

## Variables

New variables in JavaScript are declared using the **var** keyword:

var a;

var name = "John";

An important difference from other languages like Java is that in JavaScript, blocks do not have scope; only functions have scope. However, starting with ECMAScript Edition 6, let and const declarations allow you to create block-scoped variables.

## Operators

JavaScript's numeric operators are +, -, \*, / and % - which is the remainder operator. Values are assigned using =, and there are also compound assignment statements such as += and -=. These extend out to x = x *operator* y.

x += 5;

x = x + 5;

You can use ++ and -- to increment and decrement respectively. These can be used as prefix or postfix operators.

The + operator also does string concatenation:

"hello" + " world"; // "hello world"

If you add a string to a number (or other value) everything is converted in to a string first. This might catch you up:

"3" + 4 + 5; // "345"

3 + 4 + "5"; // "75"

Adding an empty string to something is a useful way of converting it.

Comparisons in JavaScript can be made using <, >, <= and >=. These work for both strings and numbers. Equality is a little less straightforward. The double-equals operator performs type coercion if you give it different types, with sometimes interesting results:

"dog" == "dog"; // true

1 == true; // true

To avoid type coercion, use the triple-equals operator:

1 === true; // false

true === true; // true

There are also != and !== operators.

JavaScript also has bitwise operations.

## Control Structure

JavaScript has a similar set of control structures to other languages in the C family.

var name = "kittens";

if (name == "puppies") {

name += "!";

} else if (name == "kittens") {

name += "!!";

} else {

name = "!" + name;

}

name == "kittens!!"

JavaScript has while loops and do-while loops

while (true) {

// an infinite loop!

}

var input;

do {

input = get\_input();

} while (inputIsNotValid(input))

JavaScript's for loop is the same as that in C and Java

for (var i = 0; i < 5; i++) {

// Will execute 5 times

}

The && and || operators use short-circuit logic, which means whether they will execute their second operand is dependent on the first. This is useful for checking for null objects before accessing their attributes:

var name = o && o.getName();

Or for setting default values:

var name = otherName || "default";

JavaScript has a ternary operator for conditional expressions:

var allowed = (age > 18) ? "yes" : "no";

The switch statement can be used for multiple branches based on a number or string:

switch (action) {

case 'draw':

drawIt();

break;

case 'eat':

eatIt();

break;

default:

doNothing();

}

## Objects

JavaScript objects can be thought of as simple collections of name-value pairs. As such, they are similar to:

* Dictionaries in Python
* Hashes in Perl and Ruby
* Hash tables in C and C++
* HashMaps in Java
* Associative arrays in PHP

There are two basic ways to create an empty object:

var obj = new Object();

And:

var obj = {};

And....

function Person(name, age) {

this.name = name;

this.age = age;

}

// Define an object

var You = new Person("You", 24); // We are creating a new person named "you" // (that was the first parameter, and the age..)

Once created, an object's properties can again be accessed in one of two ways:

obj.name = "Simon";

var name = obj.name;

And...

obj["name"] = "Simon";

var name = obj["name"];

These are also semantically equivalent. The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time though using this method prevents some JavaScript engine and minifier optimizations being applied. It can also be used to set and get properties with names that are reserved words:

obj.for = "Simon"; // Syntax error, because 'for' is a reserved word obj["for"] = "Simon"; // works fine

Object literal syntax can be used to initialize an object in its entirety:

var obj = {

name: "Carrot",

"for": "Max",

details: {

color: "orange",

size: 12

}

}

Attribute access can be chained together:

obj.details.color; // orange

obj["details"]["size"]; // 12

## Arrays

Arrays in JavaScript are actually a special type of object. They work very much like regular objects (numerical properties can naturally be accessed only using [] syntax) but they have one magic property called 'length'.

One way of creating arrays is as follows:

var a = new Array();

a[0] = "dog";

a[1] = "cat";

a[2] = "hen";

a.length; // 3

A more convenient notation is to use an array literal:

var a = ["dog", "cat", "hen"];

a.length; // 3

Note that array.length isn't necessarily the number of items in the array. Consider the following:

var a = ["dog", "cat", "hen"];

a[100] = "fox";

a.length; // 101

Remember — the length of the array is one more than the highest index.

If you query a non-existent array index, you get undefined:

typeof a[90]; // undefined

If you take the above into account, you can iterate over an array using the following:

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

// Do something with a[i]

}

This is slightly inefficient as you are looking up the length property once every loop. An improvement is this:

for (var i = 0, len = a.length; i < len; i++) { // Do something with a[i]

}

An even nicer idiom is:

for (var i = 0, item; item = a[i++];) {

// Do something with item

}

Another way to iterate is to use the for...in loop. Note that if someone added new properties to Array.prototype, they will also be iterated over by this loop:

for (var i in a) {

// Do something with a[i]

}

# Web Development Using Javascript

## Setting the development environment

You can do web development by using just as set of simple and accessible tools:

* **Plain Text Editor**: any plain text editor will work. You can find some free options below:
  + SublimeText (<http://www.sublimetext.com/>)
  + Notepad++ (<http://www.notepad-plus-plus.org/>)
  + Visual Studio Community Edition (<http://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>)
* **Browser**: one that supports javascript. Don’t be alarmed, the most popular browsers supports it: **Internet Explorer**, **Google Chrome**, **Firefox**,  **Safari**, **Opera**, etc. All of them has a javascript interpreter embedded. Since javascript is not restricted to just web development you can find a javascript interpreter in the link below:
  + Nodejs (<http://nodejs.org/>)
* **Debug Tool**: since debugging is a very important task in the development work, is really important to count on a very good tool to do it. All development platforms/IDEs offers debug capabilities. In the case of javascript and related to web development specifically you can find that browsers have embedded tools to achieve this task. Additionally you can take a look to the tool in the following link: Firebug (http://getfirebug.com/)
* **Frameworks**: basically speaking, frameworks are libraries that expose functionalities corresponding to common and recurrent task not supported natively by the language. Javascript has many frameworks available; you can find in the list below some of the most relevant (popular) ones:
  + **JQuery**
  + **Moo Tool**
  + **Angular JS**

The one described above is not the only way and not intended to be, .Net platform, java, PHP, among others, have their own approach to face web development. Here we’re only trying to show that you can do web development with what you have in your Windows PC, Mac, Linux or whatever you use, using only free access tools.

## Approach with using no frameworks

In this case, we are going to simplify the web page concept as a plain text file containing HTML code and some script code. In our scope the script code refers to Javascript code.

Remember that web browsers, basically speaking, are programs that interpret, manipulate and render html code.

In other hand, Javascript code is used to manipulate the DOM (Data Object Model) in client side and to invoke some server side functionalities by submitting forms, using AJAX and some other technologies.

HTML, AJAX and other mentioned technologies are out of the scope of this document, but you can find plenty information about on the internet.

## Approach with using Angular JS

Before to start to talk about Angular JS in property there are some concepts to deal with, first.

### Design Patterns

Since the main focus here is javascript, we only are going to overview the design patterns and explain a couple of patterns supported by Angular JS, which will be detailed later.

Design Patterns are proven solutions to recurrent problems. Therefore, an important advantage of using design pattern is to apply the industry knowledge in new projects to avoid reinvent the wheel over and over but speed up the development by applying these known solutions. It is important to highlight that patterns are concepts and in this way, they can be implemented in various languages, not only Javascript.

Since Javascript is a language mainly focused in presentation layer (front end) two design patterns focused in the front end will be examined: **Model-View-Controller (MVC)** and **Model-View-View Model (MVVM)**. Both of them pursuits the same goal, that is **Separations of Concerns**. This patterns look for applying the **Single Responsibility Principle** that says: each system’s component must have one and only one responsibility; in other words, be dedicated to do only one thing. Breaking this architectural principle makes your code less readable and harder to maintain. Said that, let’s talk first about common components for both patterns:

**Model**: Contains the domain data to be displayed. These data is independent from user interface. In both patterns the model only knows itself; it doesn’t know about the other components and that’s because it doesn’t need it.

**View**: This pattern’s component takes care about display data. Tough both patterns contains a view, the way it interacts with the others components is different. The explanation for these interactions will be deferred to description for specific pattern components.

### MVC’s Specific Components

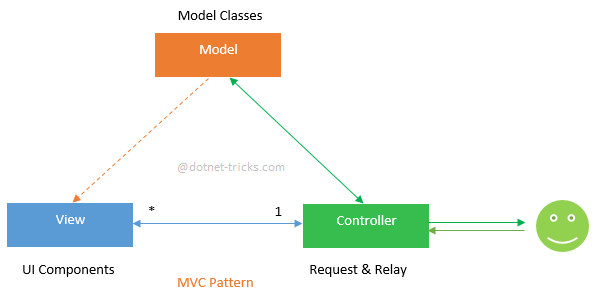
**Controller:** This component put all pieces together in the pattern. The controller may send commands to the model in order to modify their state. Controller can also send commands to the view associated to model to modify their presentation. The controller is also responsible to process incoming requests from users via the view; after process these requests and often after change the state of the model during this processing, returns the modified model to the view to be displayed, so the user can the see the result of their request. As you can notice, controllers act as a coordinator between view and model activities.

### Views and Models in the MVC pattern

Though this pattern increases separation of concerns between backend and frontend code, it maintains certain dependencies between the components inside the pattern.

The view must know what model it will displays and requires some logic to display the model properly, but this is just display logic ideally. Since the view knows the view it display, and it can only displays one and only one view, exists a coupling (dependency) of the view respect to its associated model; however, model has no dependencies for it only knows about itself. Therefore a view displays only a model, but a single model can be displayed by multiple views.

In the MVC pattern, the changes in the view and the model are manually performed for the controller. Since view’s responsibility consists in display data to the user and receive request from he/she, view may admits logic, but this code is for display purpose only.



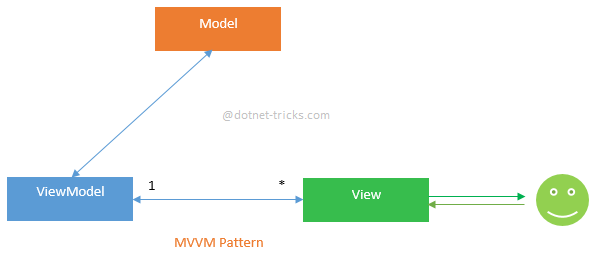
### MVVM’s Specific Components

**View Model:** this component acts a mediator between the others components in the pattern: that is the view and the model. View Models however, work in a very different way than controllers do. Here the coupling still existing in the MVC pattern is removed, increasing the ability to test components independently but with a slight sacrifice in performance due to binding mechanism.

What View Models do is expose public properties, commands and methods that help to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself. As a consequence, the view model maintains the view and the model synchronized throughout bindings. Hence, the binder is a very important piece in the view model component. It allows define, in a declarative fashion, data bindings between fields in the view and properties in the model. These bindings may be either, one-way or two-ways. A two-way binding means that whenever the view changes, the bound property in model changes automatically and likewise, whenever a bound property in the model changes triggers a change to its related field in the view. One-way binding, as its name indicates uses the same logic than two-ways binding, but in a single direction.

At this point you can see that in the MVVM pattern views do not know anything about the model and vice versa. Precisely is because of this that you’re able to test components in the pattern independently.

As you can see in the collaboration diagram, in MVVM the view is active, that means it interacts directly with the user, in clear difference than its counterpart in the MVC pattern, which is passive and requires the controller to take care of interactions with the user.



## What Is Angular?

## **AngularJS** is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's data binding and dependency injection eliminate much of the code you would otherwise have to write. And it all happens within the browser, making it an ideal partner with any server technology.

### The Angular Spot

Angular simplifies application development by presenting a higher level of abstraction to the developer. Like any abstraction, it comes at a cost of flexibility. In other words, not every app is a good fit for Angular. Angular was built with the CRUD application in mind. Luckily CRUD applications represent the majority of web applications. To understand what Angular is good at, though, it helps to understand when an app is not a good fit for Angular.

Games and GUI editors are examples of applications with intensive and tricky DOM manipulation. These kinds of apps are different from CRUD apps, and as a result are probably not a good fit for Angular. In these cases it may be better to use a library with a lower level of abstraction, such as jQuery.

### Data Binding in Angular Templates

  
Angular templates work differently. First the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. This makes testing a snap because it is easy to test your controller in isolation without the view and the related DOM/browser dependency.

### Controllers

In Angular, a Controller is a JavaScript **constructor function** that is used to augment the [Angular Scope](https://docs.angularjs.org/guide/scope).

When a Controller is attached to the DOM via the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**. A new **child scope** will be available as an injectable parameter to the Controller's constructor function as $scope.

Use controllers to:

* Set up the initial state of the $scope object.
* Add behavior to the $scope object.

Do not use controllers to:

* Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. Angular has [databinding](https://docs.angularjs.org/guide/databinding) for most cases and [directives](https://docs.angularjs.org/guide/directive) to encapsulate manual DOM manipulation.
* Format input — Use [angular form controls](https://docs.angularjs.org/guide/forms) instead.
* Filter output — Use [angular filters](https://docs.angularjs.org/guide/filter) instead.
* Share code or state across controllers — Use [angular services](https://docs.angularjs.org/guide/services) instead.
* Manage the life-cycle of other components (for example, to create service instances).

Eje:

var myApp = angular.module('myApp',[]);

myApp.controller(‘MyFirtsController', ['$scope', function($scope) {

$scope.greeting = 'Hola!';

}]);

HTML

<html ng-app="myApp">

<div ng-controller="MyFirtsController">

<p>The food is {{greeting}} spicy!</p>

</div>

</html>

### Templates

In a simple app, the template consists of HTML, CSS, and Angular directives contained in just one HTML file (usually index.html).

In Angular, templates are written with HTML that contains Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

These are the types of Angular elements and attributes you can use:

* [Directive](https://docs.angularjs.org/guide/directive) — an attribute or element that augments an existing DOM element or represents a reusable DOM component.
* [Markup](https://docs.angularjs.org/api/ng/service/$interpolate) — the double curly brace notation {{ }} to bind expressions to elements is built-in Angular markup.
* [Filter](https://docs.angularjs.org/guide/filter) — formats data for display.
* [Form controls](https://docs.angularjs.org/guide/forms) — Validates user input.

The following code snippet shows a template with [directives](https://docs.angularjs.org/guide/directive) and curly-brace [expression](https://docs.angularjs.org/guide/expression) bindings:

<html ng-app>

<!-- Body tag augmented with ngController directive -->

<body ng-controller="MyController">

<input ng-model="foo" value="bar">

<!-- Button tag with ng-click directive, and

string expression 'buttonText'

wrapped in "{{ }}" markup -->

<button ng-click="changeFoo()">{{buttonText}}</button>

<script src="angular.js">

</body>

</html>

### Filters

A filter formats the value of an expression for display to the user. They can be used in view templates, controllers or services and it is easy to define your own filter.

The underlying API is the [filterProvider](https://docs.angularjs.org/api/ng/provider/$filterProvider).

Filters can be applied to expressions in view templates using the following syntax:

{{ expression | filter }}

E.g. the markup {{ 12 | currency }} formats the number 12 as a currency using the [currency](https://docs.angularjs.org/api/ng/filter/currency) filter. The resulting value is $12.00.

Filters can be applied to the result of another filter. This is called "chaining" and uses the following syntax:

{{ expression | filter1 | filter2 | ... }}

Filters may have arguments. The syntax for this is

{{ expression | filter:argument1:argument2:... }}

E.g. the markup {{ 1234 | number:2 }} formats the number 1234 with 2 decimal points using the [number](https://docs.angularjs.org/api/ng/filter/number) filter. The resulting value is 1,234.00.

### What are Scopes?

[scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) is an object that refers to the application model. It is an execution context for [expressions](https://docs.angularjs.org/guide/expression). Scopes are arranged in hierarchical structure which mimics the DOM structure of the application. Scopes can watch [expressions](https://docs.angularjs.org/guide/expression) and propagate events.

### Scope as Data-Model

Scope is the glue between application controller and the view. During the template [linking](https://docs.angularjs.org/guide/compiler) phase the [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) set up [$watch](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch)expressions on the scope. The $watch allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM.

Both controllers and directives have reference to the scope, but not to each other. This arrangement isolates the controller from the directive as well as from the DOM. This is an important point since it makes the controllers view agnostic, which greatly improves the testing story of the applications.

angular.module('scopeExample', [])

.controller('MyController', ['$scope', function($scope) {

$scope.username = 'World';

$scope.sayHello = function() {

$scope.greeting = 'Hello ' + $scope.username + '!';

};

}]);

### Services

Angular services are substitutable objects that are wired together using [dependency injection (DI)](https://docs.angularjs.org/guide/di). You can use services to organize and share code across your app.

Angular services are:

* Lazily instantiated – Angular only instantiates a service when an application component depends on it.
* Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

## Using a Service

To use an Angular service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. Angular's [dependency injection](https://docs.angularjs.org/guide/di) subsystem takes care of the rest.

angular.

module('myServiceModule', []).

controller('MyController', ['$scope','notify', function ($scope, notify) {

$scope.callNotify = function(msg) {

notify(msg);

};

}]).

factory('notify', ['$window', function(win) {

var msgs = [];

return function(msg) {

msgs.push(msg);

if (msgs.length == 3) {

win.alert(msgs.join("\n"));

msgs = [];

}

};

}]);

### What is a Module?

You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.

### Why?

Most applications have a main method that instantiates and wires together the different parts of the application.

Angular apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped. There are several advantages to this approach:

* The declarative process is easier to understand.
* You can package code as reusable modules.
* The modules can be loaded in any order (or even in parallel) because modules delay execution.
* Unit tests only have to load relevant modules, which keeps them fast.
* End-to-end tests can use modules to override configuration.

# References

* Javascript genral concepts
  + [https://developer.mozilla.org/en-US/docs/Web/JavaScript/A\_re-introduction\_to\_JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript?redirectlocale=en-US&redirectslug=JavaScript%2FA_re-introduction_to_JavaScript)
  + <https://msdn.microsoft.com/es-co/library/ie/d1et7k7c%28v=vs.94%29.aspx>
  + <https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia>
  + <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
* Design Patterns:
  + Wikipedia
  + Dotnet Tricks.com
  + http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained
  + http://www.codeproject.com/Articles/228214/Understanding-Basics-of-UI-Design-Pattern-MVC-MVP
* Angular JS: <https://docs.angularjs.org/guide>