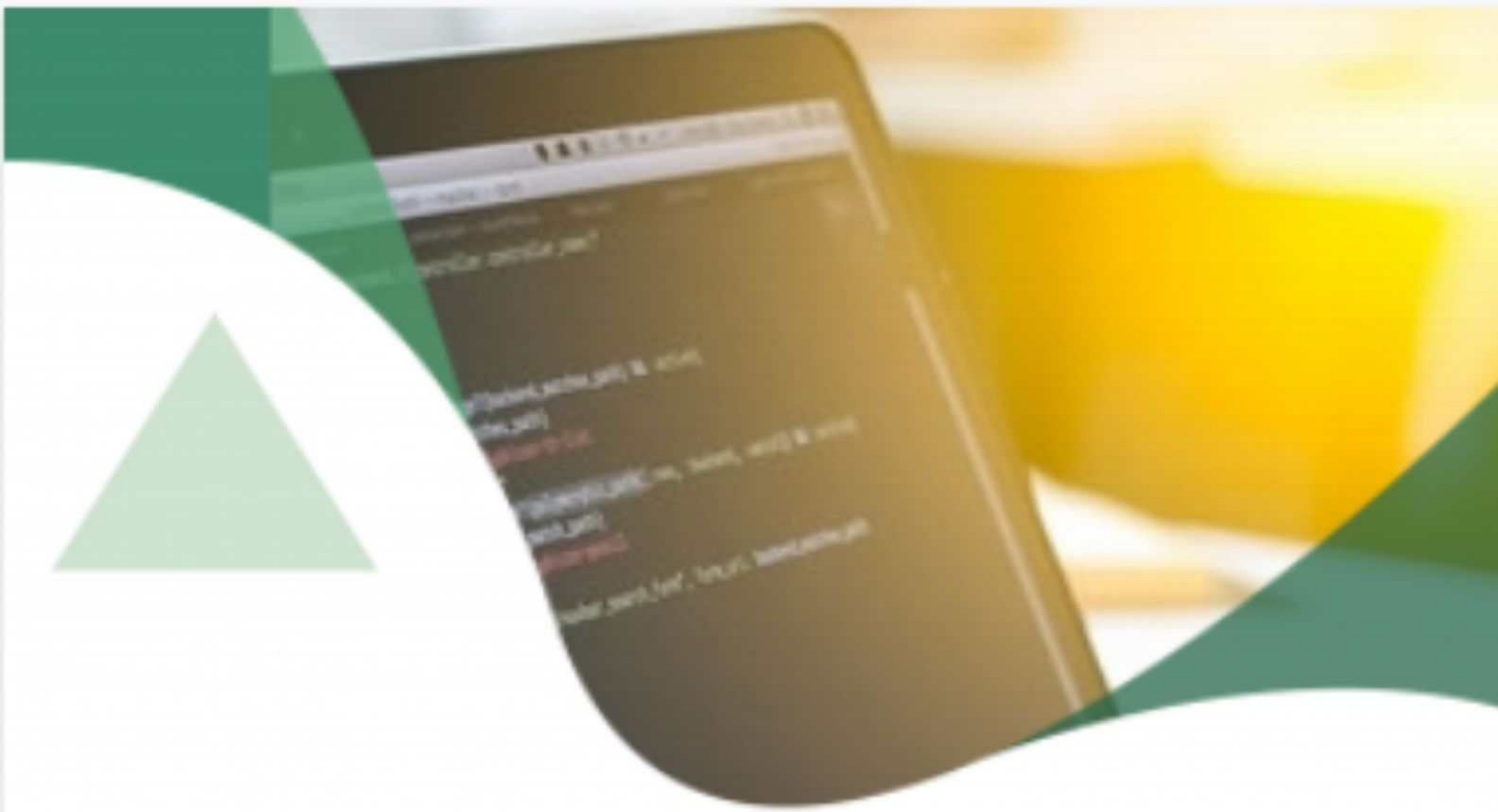

CODING WITH JAVASCRIPT FOR DUMMIES

EVERYTHING
YOU NEED TO KNOW ABOUT
JAVASCRIPT PROGRAMMING
1ST EDITION - 2020 / 7 / 14

PENGUIN RANDOM HOUSE LLC

1ST EDITION
BY CLAUDIA ALVES

Coding
with JavaScript
For Dummies

Everything you need to know about Javascript programming

1st edition
2020

By Claudia Alves

"Programming isn't about what you know; it's about what you can figure out." ***- Chris Pine***

Memlnc.com

INTRODUCTION

JAVASCRIPT IS EVERYWHERE
JAVASCRIPT ON THE WEB
JAVASCRIPT IN THE BACKEND
JAVASCRIPT AS LANGUAGE FOR DEVICE APPS
JAVASCRIPT AS A LANGUAGE FOR DESKTOP APPLICATIONS
JAVASCRIPT ON THE WEB
WEBSITES
SINGLE PAGE APPLICATIONS
LIBRARIES, FRAMEWORKS AND DEVELOPMENT "VANILLA JAVASCRIPT" FOR THE WEB
HTML5 APIS
WEB COMPONENTS
JAVASCRIPT AS THE FIRST LANGUAGE
EASY TO USE
WIDE APPLICATION AREAS
IT IS AN OPEN AND STANDARD LANGUAGE FOR THE WEB
MULTIPLE PROFESSIONS START WITH JAVASCRIPT
FRONTEND DEVELOPER:
BACKEND DEVELOPER
FULLSTACK DEVELOPER
HOW TO LEARN JAVASCRIPT
LEARN JAVASCRIPT AND NOT A DERIVATIVE
KNOWLEDGE OF THE WEB PLATFORM
DIFFERENCES BETWEEN JAVA AND JAVASCRIPT
OPEN LANGUAGE / PROPRIETARY LANGUAGE
BEFORE STARTING
USES OF JAVASCRIPT

What do you need to work with Javascript
Recommended prior knowledge

CHAPTER I

Javascript language versions
The evolution of Javascript
ECMASCript Standard
HTML 5 APIs
Language versions
Conclusion to Javascript versions and compatibility

CHAPTER II

Open a child window
A welcome message
Current date
Back button
Javascript language
Javascript is written to the HTML document
The placement of the scripts does matter
Ways to write Javascript scripts
Direct execution of Javascript code
Running javascript in response to an event
Hide Javascript scripts in old browsers
Hide Javascript code with HTML comments
Show a message for old browsers with <NOSCRIPT>
More about scripting
Interesting methods of placing Javascript scripts
Indicate the language we are using
Include external Javascript files
The Importance of Using External Javascript Code Files

CHAPTER III

JAVASCRIPT SYNTAX

Comments in the code
Upper case and lower case
Separation of instructions
Variables in Javascript
Variable concept
Rules for variable naming in Javascript
Variable names in Javascript are case sensitive
Declaration of variables in Javascript
Variable declaration with var
Declaration of Javascript variables with let and const
Scope of variables in Javascript
Variable scope concept
Global variables
Local variables
Differences between declaring variables with var, or not declaring them
Variable declaration with let
What can we save in variables
Data types in Javascript
Numeric data type
Boolean type
Data type character string
Escape characters in text strings

CHAPTER IV

JAVASCRIPT OPERATORS

Examples of using operators

Arithmetic operators
Assignment operators
Examples
Chain operators
Logical operators
Conditional operators
Bit-level operators
Operators precedence
Javascript typeof operator for type control
Recommendations with typeof
Javascript control structures
Decision making
IF structure in Javascript
Conditional expressions
Nested IF statements
IF operator
Javascript SWITCH structure
FOR loop in Javascript
Sample for-loop exercise
WHILE and DO WHILE loops
WHILE loop
DO ... WHILE loop
Example of the use of while loops
Break and continue
Break
Continue
Additional example of the break statement
Nested loops in Javascript
Javascript functions
What is a function
How to write a function
How to call a function

Where do we put the Javascript functions
Function parameters
Parameters
Multiple parameters
Parameters are passed by value
Return values in Javascript functions
Returning values in functions
Multiple return
Scope of variables in functions
Javascript function library
Eval function
ParseInt function
Arrays in Javascript
Creation of javascript arrays
Data types in arrays
Array declaration and summary initialization
Array length
Multidimensional arrays in Javascript
Array initialization
General introduction to objects in Javascript
What is an object
How to access object properties and methods
How to create objects
Create and instantiate objects from functions
• Object.freeze
Freeze objects
Freeze matrices
Deep freezing
undefined in JavaScript
Null in JavaScript

WORD PROCESSING

Smuggling of characters
Long text strings
Access the characters
Compare text strings
• String.fromCharCode
Handle values larger than the maximum
• String.fromCodePoint
String.raw ()

Introduction

Getting started is easy, but there is a long way to go if you want to express all the possibilities of language. Javascript is one of the standard languages of the web and therefore ideal for many of the professionals who plan to dedicate themselves to this medium. But even if you don't want to develop specifically for the web, Javascript is an excellent alternative for making mobile applications or desktop applications.

Perhaps not all developers need to reach an advanced level of Javascript, but a basic knowledge will certainly be of great help at many points in their professional careers. In any case, once the learning is completed, you will have endless opportunities at your fingertips.

We'll start by looking at the state of Javascript today (2017) and the reasons why Javascript is worth using in general. Then we will take care of explaining how you can learn Javascript and get to whatever level we set ourselves.

Javascript is everywhere

Javascript has become an ideal language to learn, since it has many and varied applications, in addition to providing simplicity for people who are starting out. To run it we only need a browser, although at the moment Javascript has surpassed the scope of web clients, to be located almost anywhere.

Javascript on the web

The environment where Javascript first appeared was the web. Its execution focused on the scope of a web page and allowed developers to provide interactivity, manipulate the document or the browser window, perform calculations, etc. Netscape Navigator, a browser that disappeared today, has the honor of having introduced Javascript as a language, although today it is supported by all web clients with which a user can navigate.

Using Javascript for the web, in the browser environment, is also known as client-side programming. Today it is still the most common environment for Javascript execution, but it really is just one more among its possibilities.

Javascript in the backend

Certain developers came up with the idea that they could extract the Javascript runtime, which until then had only been available within the scope of the browser, to use it for any other purpose outside the web client. This is how NodeJS was born, which is nothing more than Javascript outside the browser.

With NodeJS we can program with Javascript applications that run directly on the operating system and that are capable of solving any type of problem. With this technology, Javascript became a general-purpose programming language.

One of the most common uses of NodeJS is backend programming. It allows you to program applications that are capable of running on the server, providing access to databases, the file system, and any other server-side resources. However, NodeJS is so comprehensive that it can be used for many other tasks, such as automation, optimization, or application deployment, among other operations.

Javascript as language for device apps

For years it has also been possible to use Javascript as a language for creating applications for devices (mobile apps, tablets, TVs ...). The applications developed are installed from the corresponding app stores of the main mobile systems and the user in principle does not perceive any difference from these with respect to the applications developed with the native languages. But, due to the fact that they are programmed with Javascript and HTML5, they open a new field of action for people experienced in developing for the web and dispense with the learning of native languages for each mobile platform.

Initially, to run applications made in Javascript and HTML5, a "Web View" was required. Basically the function of the web view is to offer a framework for the execution of the app, so that it runs within a browser, even if the user does not perceive it. This situation has various advantages and disadvantages that we will not go into, but today there is also the possibility of using Javascript as a development language for native applications, which do not require a web view to function.

Examples of frameworks for the development of applications based on web view we have Apache Cordova, Phone Gap or Ionic. Examples of frameworks for developing native applications using Javascript include Native Script and React Native. All the alternatives have the important advantage of producing apps for Android and iOS with the same code base, as well as for other minority systems.

Javascript as a language for desktop applications

Another area in which Javascript has penetrated strongly is in the development of applications for personal computers. With Javascript we are able to create advanced applications, capable of using all the resources of a computer and also run on any operating system we need.

Using Javascript for desktop applications is easy thanks to projects such as Electron, which allows us to produce cross-platform applications, that is, they can be installed on Windows, Mac OS X and Linux. There are quite a few well-known applications developed with Electron, such as Atom, Visual Studio Code, Slack, Hyper, etc.

Javascript on the web

The field of action of Javascript is very wide, providing alternatives for almost any execution environment. However, if we want to start learning Javascript, the most suitable place is the web.

Within the discipline of client-side development (Javascript executed within the browser environment) we can find different types of projects, which also require different approaches and knowledge.

Websites

When we refer to websites here we mean sites where the most important part is content, be they blogs, news pages, and even e-commerce.

Javascript in these cases is dedicated to providing functionality and interaction, allowing for dynamic user interfaces, response to user actions, form validation, etc.

Single Page Applications

In recent years, the web has become popular as a platform for business applications. Applications called "management", which were previously run with desktop programs, today have web fronts that allow us to use them from the cloud, that is, from any browser connected to the Internet and without the need to install software on the machine. In this type of application, it is common to carry a large part of the processing load from the server side to the client. In this new paradigm, the browser is responsible for doing many more things than in traditional websites, such as creating the HTML code to visualize the data or navigating between screens or application routes.

In Single Page Applications, also known as SPA, it is normal for the server to deliver only the raw business data and for the browser to do all the work of presenting that data in a suitable format (HTML is produced in the browser to represent that data). But what most characterizes a SPA is that navigation is always done within the same page and Javascript is in charge of presenting one or another screen to the user without having to reload the entire page.

The two main factors that characterize SPAs, 1) the fact of bringing raw data from the server (lighter) and 2) all navigation is done within the same document, they produce web applications with a very fast response, providing an experience of use close to that of a desktop application.

Libraries, frameworks and development "Vanilla Javascript" for the web

The creation of a SPA is a much more advanced task than the development of a website and in order to carry out this work it is important that the development team is based on a Javascript framework, such as Angular, React, VueJS, Ember or Polymer. .

Note: React and Polymer are considered more libraries, but with a series of add-ons, which they themselves provide in many cases, they offer as many features as those found in a framework. In case someone does not know yet, a library is a set of functions, or classes and objects, that allows us to perform a range of common tasks for the development of certain application needs. A framework is mainly distinguished from a library because, in addition to providing code to solve common problems, it offers an architecture that developers must follow to produce applications and ensure better code quality and greater ease of maintenance. In other words, the framework, in addition to offering diverse utilities, marks a style and workflow when developing applications.

For the development of websites, it would generally be sufficient to use pure Javascript, without the need to rely on any additional library. That "pure" Javascript development is generally known as "Vanilla Javascript".

Note: It should be clear that "Vanilla" is not a trademark or any flavor of Javascript beyond that of the language itself. It's like a joke to indicate that with Javascript (and nothing else) you can solve all the things that libraries and ready-made frameworks offer you.

However, it is also common for libraries such as jQuery to be used in website

development. jQuery is a set of objects and functions (general utility code) that aims to help you manipulate the page, saving the differences between different browsers and allowing you to write a single code that runs correctly in any web client. In addition, jQuery offers you many functions that are really useful for the development of many common website tasks, which you can use faster than if you only work with Javascript.

Note: Regarding jQuery, it is worth mentioning that there is a current of developers who warn that using jQuery is not absolutely necessary in most cases. Using jQuery is not bad, but many people implement it to solve needs that a little bit of Javascript "Vanilla" is capable of. Sometimes jQuery is loaded by default, perhaps for convenience, to get carried away or simply due to ignorance of the Javascript language itself, and yet very few of its functions are used.

There are more specialized libraries to solve each and every one of the things that a general library like jQuery offers and that take up much less download weight and processing time for browsers.

But beyond libraries and frameworks we must know that today, in addition to the language itself, there are many other Javascript-based technologies in browsers to solve a wide range of needs.

HTML5 APIs

With the advent of HTML5, there was a greater standardization of browsers, reaching a commitment by manufacturers to support web languages (HTML + CSS + Javascript) as dictated by their specifications. But it also produced an abundant stream of new specifications to work with the widest range of browser, computer, or device resources.

HTML5 offers APIs for working with elements such as the camera, geolocation, storage, bitmap or vector drawing, audio, video, etc. Everything HTML5 offers is available in all browsers and is part of the Javascript developer's toolkit, without using any library or framework.

Web Components

The next development revolution for the Internet, which has not yet exploited in the middle of 2017, is called Web Components. It is based on a new API (with several specifications together) aimed at creating components. Components are like new HTML tags that any developer can create to solve common or particular application problems.

With web components developers can extend HTML by creating new components capable of doing anything and with advanced encapsulation capabilities, to respect their autonomy and capable of being used in any project, maximizing code reuse capabilities without the need to be based on any type of library or framework.

Like HTML5, Web Components are part of the possibilities offered by browsers by default, although their support right now is not as universal.

Note: In the coming months, all browsers are expected to support Web Components V1 and we can use it without any restrictions. Meanwhile there is a Polyfill that allows to extend the support to Web Components to browsers that do not yet have it. Polyfills are literally "gap fillers", which make up for the deficiencies of old browsers with respect to the standards. In the HTML5 APIs they have already begun to be used intensively to allow the use of new features of the web languages in old or slightly updated clients.

Javascript as the first language

If you are learning programming, Javascript is an excellent bet to start. Basically for three main reasons:

Easy to use

We only need a browser to be able to execute Javascript. It does not require any kind of compilation (process to create a binary executable file for a particular operating system) but is interpreted. This involves a more streamlined workflow, making the first steps easy. In addition, Javascript is dynamic typing and its syntax is less elaborate than that of other languages and allows things to be done in different ways, according to the skill, preferences or customs of each programmer.

Wide application areas

As we have seen, Javascript has practically unlimited uses. It means that you can use the language practically for what you need. By learning a single language you will be able to reach any purpose you set for yourself.

It is an open and standard language for the web:

At some point in their professional activity, most developers will work on the web environment. Learning programming with Javascript assures us that the knowledge will be applied directly at various points in the processional career of students.

These are the reasons why we have been teaching how to program with Javascript for more than 10 years and why in 2017 the prestigious universities of Stanford have abandoned their introductory course to programming with Java in favor of Javascript.

Multiple professions start with Javascript

Another reason why it is worth starting with Javascript is because language is one of the fundamental pillars to perform multiple professions in demand within the scope of the web.

Frontend developer:

He is the professional who deals with client-side development, although he can have various activities in addition to programming, such as design, layout, user interface development, client-side SPA applications, etc. Javascript knowledge is essential in all areas of action of a frontend developer.

Backend developer:

As a backend we understand server-side programming. For backend, the truth is that Javascript (with NodeJS) is just one of the many possibilities. However, NodeJS offers many advantages such as its asynchrony, speed and optimization, making it ideal for many types of projects.

Fullstack developer:

Fullstack developer is one that is capable of working on both the client and server side. In reality it is a rare breed, so much that it is often said that it does not really exist, because it requires a lot of knowledge from the developer and therefore it is very complicated, or impossible, that a single profile brings them all together. However, in my opinion it is a reality in many jobs since there are professionals who must have global knowledge in various areas (autonomous freelancers are the main example) and therefore it is a figure that really does exist. Today, the fact of Javascript serving both the client and the server side, makes the figure of the fullstack developer much easier and makes life easier for thousands of developers, as it saves them the mental fatigue of moving from language to language. another constantly.

Note: Specialization is also present in these professions. It is truly amazing to see how profiles develop in the world of the web and how new specialized professions are generated year after year. What we used to call frontend, simply, today we can subdivide it into dozens of specific profiles or professions such as "frontend engineer", "frontend web designer", "CSS architect", "mobile frontend developer", "frontend devops" ...

How to learn Javascript

Learning Javascript at an advanced level takes months of study, but getting started is easy and in no time you will be able to do amazing things with little effort. In the Javascript manual you will be able to know the language, but to start we want to give you a couple of tips.

Learn Javascript and not a derivative

Our first tip if you are just starting out is to learn Javascript and not a library or framework. In the end, all the projects on the web use Javascript and everything you can do with a library or framework can also be done with Javascript, so the correct thing to do is start by mastering the language, and then set new objectives in the medium or long term.

In short, don't learn jQuery: learn Javascript. Do not learn Angular: learn Javascript, do not learn React: learn Javascript ... With a solid base of Javascript it will be much easier for you to learn later any library or framework on which you want to base. Also, you will not have problems in the future, when you want to do things for which that library is not intended or you want to customize any detail of your application.

What should accompany the knowledge and use of Javascript are your skills in everything that the browser (the web platform) offers you by default: HTML5 and Web Components. Everything you do based on Javascript "Vanilla", HTML5 and Web Components you have the certainty that it can be used in any project where you can get to work, regardless of the library or framework that is being used in each case.

Knowledge of the web platform

Our second advice is that, if you are going to work in this medium, you must bear in mind each of the peculiarities of the web platform. There is a lot of general knowledge that you need to acquire that will give you a foundation on which to build your skills in the world of web development.

Knowledge of the medium: You must know what the Internet is, the Web, the HTTP protocol, the domain name system and of course the fundamental languages to specify the content and form: HTML and CSS.

Design knowledge: Although your profile may be more of a programmer, it is ideal to have a vision, at least technical, of the design characteristics for the web. User experience, usability, general graphic design or accessibility are important points.

Programming knowledge: If you are going to dedicate yourself to the programming profession it is not only enough to have a basic knowledge of code and structured programming, it is ideal to be interested in object-oriented programming, analysis and design of software, design patterns, bases data, etc.

Knowledge of tools: The professional must also know a large group of tools for day-to-day work, from the command line terminal and basic administration of servers, to the automation of tasks, through the version control tools (Git preference) and optimization.

Differences between Java and Javascript

We are counting various interesting issues and curiosities that serve as an introduction to the Javascript Manual and we want to discuss one of the most typical associations that are made when hearing about Javascript. We mean to relate it to another programming language, called Java, which doesn't have much to do with it.

Javascript was really called that because Netscape (the browser that launched Javascript), which was allied with the creators of Java at the time (Sun Microsystems), wanted to take advantage of the brand image and popularity of Java, as a maneuver for the expansion of its new language. All in all, a product was created that had certain similarities, such as language or name syntax. He was made to understand that he was a little brother and

specifically oriented to do things on web pages, but also made many people fall into the error of thinking that they are the same.

We want to make it clear that Javascript has nothing to do with Java, except for the collaboration agreement of the creators of both languages, as it has been read. They have always been totally different products that bear no more relation to each other than the identical syntax, although in truth, the syntax of both is inherited from another popular language called C.

Differences between Java and Javascript

Most notable differences between Java and Javascript

Some of the most representative differences between these two languages are the following:

Compiler. To program in Java we need a Development Kit and a compiler. However, Javascript is not a language that requires your programs to be compiled, but rather they are interpreted by the browser when it reads the page.

Object oriented. Java is a oriented programming language

to objects. (Later we will see what it means object-oriented, for which you do not know yet). Javascript is a "multi-paradigm" language, it does not require object-oriented programming, although it does allow it. This means that we will be able to program in Javascrip without creating classes, as is done in structured programming languages such as C or Pascal.

Purpose. Java in principle is much more powerful than Javascript, because it is a general-purpose language. With Java you can make applications of the most varied, however, with Javascript we can only write programs to run on web pages. Although to tell the truth, since the last decade Javascript has expanded so much that today we can use the language to make applications of all kinds, such as console programs, mobile applications, desktop applications, etc. Therefore, today it is Javacript we could say that it is almost as powerful, or more, than Java is.

Static typing. Java is a strongly typed programming language (also called static typing). This means that when declaring a variable in Java we will have to indicate its type and it will not be able to change from one type to another throughout the execution of the program. For its part, Javascript does not have this feature, but is a dynamic typing language (or slightly typed) and we can put the information we want in a variable, regardless of its type. In addition, we can change the data type of a variable when we want.

Open language / proprietary language

. Another important difference is that Javascript is based on an open standard, which does not have a particular owner and therefore any manufacturer can freely implement it on their systems. However Java is a language owned by a company (currently Oracle), so it is directed with a particular and commercial approach.

Other features. As we see, Java is much more complex, but also more powerful and robust. Initially, Java has more functionalities than Javascript and requires much more intense learning to master it. Javascript makes learning easy, even for people with no programming experience, and allows you to make programs quickly, obtaining quite attractive results with little code and effort.

The differences that separate Java from Javascript are therefore remarkable. They are also used for very different things. Java is more thought to make complex applications, oriented to the business world, or applications for Android phones. Javascript is intended to make applications heavier, mainly web-oriented.

We want to point out that currently the Javascript web approach does not have to be considered unique. As we have said, it is possible to make almost any type of program using Javascript and in recent years it has been occupying more and more plots. Also, as new versions of the Javascript standard have been introduced, it has become more robust and suitable also for large applications.

Before starting

There are several points that we want to comment on as an introduction to the Javascript Manual and that you may want to know before starting to program. First, it would be good to get a more concrete idea of the possible applications that the language could have and that can be found on innumerable websites. In addition we also want to discuss the tools and prior knowledge we need to get down to work.

Uses of Javascript

Perhaps today it is superfluous to say what Javascript is for, but let's briefly see some uses of this language that we can find on the web to get an idea of the possibilities it has.

Without going any further, DesarrolloWeb.com uses Javascript for the top menu, which shows different links within each main option. We are changing the page from time to time, but in the current design of this website, elements such as the "Login" box also have their dynamism with Javascript.

Currently almost all the slightly advanced pages use Javascript, as it has become one of the insignia of what is called Web 2.0 and the rich user experience. For example, popular websites like Facebook, Twitter or Youtube use Javascript in abundance. To be more specific, when we click on a link on the social network to comment on something, a small form is displayed on the page that appears as if by magic and is then sent without leaving the page itself. Also when we vote for a video on YouTube or when the characters that we have written in the Twitter mini-posts are counted, Javascript is used to perform small functionalities that cannot be done with HTML alone. You can actually see Javascript examples inside any slightly complex page. Some that we will have seen countless times are dynamic calendars to select dates, calculators or currency converters, rich text editors, dynamic browsers, etc.

It is much more common to find Javascript to perform simple effects on web pages, or not so simple, such as dynamic browsers, opening secondary windows, form validation, etc. We dare to say that this language is really useful in these cases, since these typical effects have just enough complexity to be implemented in a matter of minutes without the possibility of errors. However, apart from those simple examples, we can find many applications on the Internet that base part of their operation on Javascript, which make a web page become a true interactive program for managing any resource. Clear examples are online office applications, such as Google Docs, Office

Online or Google Calendar.

What do you need to work with Javascript

To program in Javascript we need basically the same as to develop web pages with HTML. A text editor and a browser compatible with Javascript. Any minimally current computer has everything it needs to be able to program in Javascrip. For example, a Windows user has within their typical installation of the operating system, a text editor, Notepad, and a browser: Internet Explorer.

Recommended prior knowledge

The truth is that it does not take a lot of knowledge base to start programming in Javascript. Most likely, if you read these lines you already know everything you need to work, since you will have already had some relationship with the development of websites and you will have detected that to do certain things it is good to know a little Javascript.

However, it would be nice to have an advanced command of HTML, at least enough to write code in that language without having to think about what you are doing. An average knowledge of CSS and perhaps some previous experience on some programming language will also be useful, although in this manual of DesarrolloWeb.com we will try to explain Javascript even for people who have never programmed.

Chapter I

Browser and Javascript versions

To continue with the introduction to Javascript, it is also appropriate to introduce the different versions of Javascript that exist and that have evolved along with the versions of browsers. Language has been advancing during its years of life and increasing its capabilities.

This is an interesting knowledge, since when we develop with Javascript we depend directly on the execution platform and the compatibility or not with modern language alternatives.

Already to warn you from the outset, in this Javascript Manual we are going to work with fully extended versions of the language in all browsers, so anything we explain can be applied without any problem.

Javascript language versions

The evolution of Javascript

In the article of the introduction to Javascript we already explained some of the history of the language. We saw that in the beginning its objectives were simple and that nowadays with Javascript we can manage to create really complex pages, user interfaces and effects. Therefore, as the demands of the developers grew, so did the language itself.

It is important to mention that the language has evolved in two ways:

On the one hand, the language itself has been incorporating operators, control structures, syntax rules to do repetitive things with less code. We owe these improvements in the language to the ECMASCript standard, which we will talk about immediately.

On the other hand, browsers have been incorporating new instructions, to be able to manipulate modern elements of the page, such as divisions, CSS styles and systems such as local storage, full-screen work, geolocation and a long etc. All these improvements are defined under the standards of HTML 5 and its Javascript APIs.

In any case, what should be clear is that Javascript has been improving its features as a language, while browsers have been offering greater support to advanced functionalities for the control of page elements. That is why in Javascript we generally have to be careful with the browser market and compatibility with the functionalities that we want to use.

ECMASCript Standard

Javascript as a language is standardized by the organization "ECMA International". This company is dedicated to creating standards for communication and information processing. One of its best known standards is ECMAScript.

ECMAScript is the standard definition of the Javascript language, which any client that supports Javascript must support. Over time, different versions of ECMAScript have been published, with the latest ECMAScript 2015, also known as ES6.

ES6 is supported by all current browsers except Internet Explorer. Therefore, if you support IE programs, you could not use ES6 in principle.

This is half true, since there are systems that allow the code to be translated from one version to another of the standard, although at the moment we do not want to get into these details. If you are interested, we recommend reading the Webpack Manual, which is one of the many tools that you can use to carry out this task.

HTML 5 APIs

An API is an application programming interface. Basically it indicates how the functionalities or services of a system are going to be accessed, through which instructions and what procedures.

Browsers define an API to access the resources we have to manipulate the state of the page and accord to peripherals, storage, etc.

Browser APIs are generally specified by the W3C and must be faithfully implemented by all web clients.

It happens a bit the same as with the language versions. If the browser is very old it may not have support for some APIs, so we have to be cautious. In practice more than anything with Internet Explorer and some old mobile browser.

In this first part of the manual we are not going to cover the HTML 5 APIs or access to the browser resources. This area is the subject of study in the second part, dedicated to Client-side Javascript Development.

Language versions

In the first part of this manual the objective is to learn Javascript well, so we will limit ourselves to the study of language in particular. In this sense we will use ES5, which is the version of the language that works in all current browsers.

As general information, we will summarize we will comment on the different versions of Javascript:

Really, any moderately modern browser will now have all the Javascript functionalities that we are going to need and above all, those that we can use in our first steps with the language. However, it may be useful to know the first versions of Javascript that we discussed in this article, as a curiosity.

Javascript 1: It was born with Netscape 2.0 and it supported a large number of instructions and functions, almost all that exist now have already been introduced in the first standard.

Javascript 1.1: It is the Javascript version that was designed with the arrival of 3.0 browsers. It implemented little more than its previous version, such as dynamically processing images and creating arrays.

Javascript 1.2: The version of browsers 4.0. This has the disadvantage that it is a little different on Microsoft and Netscape platforms, since both browsers grew differently and were in full fight for the market.

Javascript 1.3: Version implemented by 5.0 browsers. In this version, some differences and rough edges between the two browsers have been filed.

Javascript 1.5: Current version, at the time of this writing, implemented by Netscape 6.

For its part, Microsoft has also evolved to present its 5.5 version of JScript (that's what they call the javascript used by Microsoft's browsers).

In 2009 the version of ECMAScript 5 was published. This is the version that

we are going to deal with mainly in the manual, which works in all browsers on the market.

In 2015 the sixth version of ECMAScript is launched, with a very significant number of new features. This version is fully usable, except in Internet Explorer. Its improvements are so important that all professional developers use them and, if necessary, the code is translated to support older browsers. This version is explained in the ES6 Manual.

There is currently a commitment to release one version of the ECMAScript standard per year. Therefore, there are various standards with small incremental changes, which have been incorporated at different rates between browsers.

Conclusion to Javascript versions and compatibility

It is obvious that, after writing these lines, many other versions of Javascript will be presented or will have been presented, because, as browsers are improved and HTML versions are released, new needs arise for programming dynamic elements. However, everything we are going to learn in this manual, even other much more advanced uses, is already implemented in any Javascript that exists today.

Chapter II

Quick effects with Javascript

Before diving into the matter, we can see a series of quick effects that can be programmed with Javascript, which can give us a clearer idea of the capabilities and power of the language. Below we will see several examples, which we have highlighted for this introduction in the Javascript Manual, for having a minimum of complexity and although they are very basic, they will come in handy to have a more accurate idea of what Javascript is when it comes to going through the following chapters.

Open a child window

First let's see that with a javascript line we can do quite attractive things. For example we can see how to open a secondary window without menu bars that shows the Google search engine. The code would be the following.

```
<script>
window.open ("http://www.google.com", "", "width = 550, height = 420, menubar = no")
</script>
```

A welcome message

We may display a pop-up text box upon completion of loading the cover of our website, which might welcome visitors.

```
<script>
window.alert ("Welcome to my website. Thanks ...")
</script>
```

Current date

Now let's look at a simple script to show today's date. Sometimes it is very interesting to show it on the websites to give an effect that the page is "up to date", that is, it is updated.

<script> document.write (new Date ()) </script>

These lines should be inserted inside the body of the page where we want the last update date to appear. We can see the example running here.

Note: A detail to highlight is that the date appears in a somewhat strange format, also indicating the time and other attributes of the same, but we will learn to get exactly what we want in the correct format.

Back button

Another quick example can be seen below. It is a button to go back, like the one we have in the toolbar of the browser. Now we will see a line of code that mixes HTML and Javascript to create this button that shows the previous page in the history, if any.

<input type = button value = Back onclick = "history.go (-1)">

The button would be similar to the following, a normal button with the default appearance that the browser and operating system you use give the buttons. Below is an image of what the button would look like on my system.

As a difference with the previous examples, it should be noted that in this case the Javascript instruction is inside an HTML attribute, onclick, which indicates that this instruction has to be executed in response to the button click.

The ease with which some interesting actions can be carried out has been verified. As you can imagine, there would be many other simple Javascript samples that we reserve for later chapters.

Javascript language

Let's see how to insert the Javascript code into an HTML document. In this article we will explain some important rules and the most basic ways of executing Javascript in the context of a page.

In this part of the manual on Javascript we are going to know the most basic way of working with the language. In this article we will give the first information on how to include scripts, mixing the Javascript code itself with the HTML.

Then we will also see how code should be placed so that our website is compatible with all browsers, even those that do not support Javascript. Many ideas of how Javascript works have already been described in previous chapters, but with the aim of leaving nothing in the pipeline, we will try to grab from here all the important data of this language.

Javascript is written to the HTML document

The most important and basic thing that we can highlight at this moment is that Javascript programming is done within the HTML document itself. That is, the Javascript code, in most cases, is mixed with the HTML code itself to generate the page.

This means that we must learn to mix the two programming languages and we will quickly see that, so that these two languages can coexist without problems between them, delimiters must be included that separate the HTML tags from the Javascript instructions. These delimiters are the <SCRIPT> and </SCRIPT> tags. All the Javascript code that we put on the page has to be inserted between these two tags.

The placement of the scripts does matter

We can enter several scripts on the same page, each one that could be entered within different <SCRIPT> tags. The placement of these scripts is not irrelevant. In the beginning, with what we know so far and the scripts that we have carried out for testing, it gives us a little the same where to place them, but in certain cases this placement will be very important. In each case, and when the time comes, it will be reported accordingly.

Javascript can also be written within certain page attributes, such as the onclick attribute. These attributes are related to user actions and are called event handlers.

Below we are going to take a closer look at these two ways of writing scripts, which have as their main difference the moment in which the sentences are executed.

Ways to write Javascript scripts

Until now in the Javascript Manual we have already had the opportunity to try some simple scripts, however, we still have to learn one of the bases to be able to work with the language and that is to learn the two ways of executing Javascript code. There are two fundamental ways to run scripts on the page. The first of these ways is direct script execution, the second is execution in response to user action.

We will now explain each of these forms of execution available, but for those who want it, we also recommend watching the video on Ways to include and execute scripts.

Direct execution of Javascript code

It is the most basic method of running scripts. In this case, the instructions are included inside the <SCRIPT> tag, as we have previously commented. When the browser reads the page and finds a script, it interprets the lines of code and executes them one after the other. We call this way direct execution because when the page is read the scripts are executed directly.

```
<! DOCTYPE html>
<html lang = "is">
<head>
  <meta charset = "UTF-8">
  <meta name = "viewport" content = "width = device-width, initial-scale = 1.0">
  <meta http-equiv = "X-UA-Compatible" content = "ie = edge">
  <title> Example of direct execution </title>
</head>
<body>
    <h1> Page with Javascript </h1>
    <p> This page has a dialog box, which will be displayed as soon as the browser processes it.
</p>

    <script>
      var people = 4;
      var amountEntrance = 9.50;
      alert ('You need' + people * ticket amount + 'euros for everyone to enter the cinema');
    </script>

    <p> When the user clicks accept in the dialog box, the browser will display the full page. </p>
</body>
</html>
```

That Javascript code will execute as the page opens. When the browser, when processing the page, finds this code, it will stop reading the page to execute the Javascript script. As a result it will show in a dialog box "You need 38 euros for everyone to enter the cinema". When the user clicks the "accept" button, they will continue reading the rest of the page and displaying the content of the entire page in the browser window. Of course we will see many other examples throughout the manual.

This method will be the one we use preferably in most of the examples in this part of the Javascript Manual. In the second part of the Javascript Manual we can learn many things and among them we will see in detail the second way of executing scripts that we are going to relate below.

Running javascript in response to an event

It is the other way to run scripts, but before we see it we must talk about events. Events are actions performed by the user. Programs like Javascript are prepared to catch certain actions performed, in this case on the page, and perform actions in response. In this way, interactive programs can be carried out, since we control the movements of the user and respond to them. There are many different types of events, for example, the push of a button, the movement of the mouse, or the selection of text on the page.

The actions that we want to perform in response to an event can be indicated in many different ways, for example within the same HTML code, in attributes that are placed inside the tag that we want to respond to the user's actions. In the chapter where we saw a quick example we already verified that if we wanted a button to perform actions when it was clicked, we had to indicate them within the onclick attribute of the button.

We therefore check that Javascript code can be inserted into certain attributes of HTML tags. However, it is not the only possible method. We can also select page elements directly with Javascript and associate functions that will be executed in response to events. Although it is a bit early so that you can understand this whole process in detail, we are going to see a simple example.

```
<! DOCTYPE html>
<html lang = "is">
<head>
  <meta charset = "UTF-8">
  <meta name = "viewport" content = "width = device-width, initial-scale = 1.0">
  <meta http-equiv = "X-UA-Compatible" content = "ie = edge">
  <title> Example hover over </title>
</head>
<body>
```

```
    <h1> Javascript Example </h1>
    <span id = "honeydew"> Mouse over here </span>
    <script>
    var passes = 0;
    function announcePast () {
      passes = passes + 1;
      alert ('Mouse over' + passes + 'times');
    }
    document.getElementById ('my element'). addEventListener ('mouseenter', announcePast);
    </script>
</body>
</html>
```

This time you have the code for an entire page. In it you can find the Javascript code embedded inside the body of the page. Unlike the previous code, when the browser reads the page it doesn't execute anything. It simply memorizes the script and associates the function with the element that has been defined. Specifically, we have defined an event of type "mouseenter" on an element of the page that has an identifier "my element". This means that each time the user places the mouse pointer over the element "my element", it will execute the function "announcePasses".

As we say, perhaps it is a little early to see examples of this style, so do not worry if you have not understood everything. We are just starting the Javascript manual and you will be able to learn all this and much more little by little. So, We will see later this type of in-depth execution and the types of events that exist. To get there we still have to learn many other things from Javascript. In the next article we will show how we can hide the Javascript code for old browsers.

Hide Javascript scripts in old browsers

Throughout the previous chapters of the Javascript Manual we have already seen that the language was implemented as of Netscape 2.0 and Internet Explorer 3.0. Even for those who do not know, it is okay to say that there are browsers that work on operating systems, where you can only display text and where certain technologies and applications are not available, such as the use of images, different typographic fonts or Javascript itself. .

Thus, not all browsers that can be used by users who visit our page understand Javascript. In cases where the scripts are not interpreted, browsers assume that their code is text from the page itself and as a consequence, present the scripts in the body of the document, as if it were normal text. To prevent script text from being written to the page when browsers don't understand it, you have to hide them with HTML comments (<! - HTML comment ->). In addition, in this article we will also see how to display a message that is seen only in browsers that do not support Javascript.

Updated: At the present time we can say that almost 100% of the available browsers support Javascript, or at least they recognize script tags, so, even if it is disabled, they will not show the text of our Javascript programs. Therefore, currently it is no longer essential to perform the operation of hiding the code of the page scripts for old browsers. However, if we want to make a completely correct page, it will be useful to learn how to hide a script so that it is never displayed as text on the page.

Hide Javascript code with HTML comments

Let's look at a code example where HTML comments have been used to hide Javascript, or better said, the code the Javascript scripts.

<SCRIPT>

<! -

Javascript code

// ->

</SCRIPT>

We see that the beginning of the HTML comment is identical to how we know it in HTML, but the closing of the comment has a peculiarity, which begins with a double slash. This is because the end of the comment contains several characters that Javascript recognizes as operators and when trying to parse them it throws a syntax error message. So that Javascript does not launch an error message, this double bar is placed before the HTML comment, which is nothing more than a Javascript comment, which we will know later when we talk about syntax.

The beginning of the HTML comment is not necessary to comment it with the double bar, since Javascript understands well that it is simply intended to hide the code. A clarification at this point: if we put the two bars on this line, they would be seen in old browsers as being outside the HTML comments. <SCRIPT> tags are not understood by older browsers, therefore they are not interpreted, as they are with any unknown tag.

Show a message for old browsers with <NOSCRIPT>

There is the possibility of indicating an alternative text for browsers that do not understand Javascript, to inform them that a script should be executed in that place and that the page is not working at 100% of its capacities. We may also suggest that visitors update their browser to a language-compatible version. To do this we use the <NOSCRIPT> tag and between this tag and its corresponding closing we can place the alternative text to the script.

<SCRIPT>

javascript code

</SCRIPT>

<NOSCRIPT>

This browser does not understand the scripts that are running, you must update your browser version to a newer one.

**

**

** Netscape .
**

** Microsoft .**

</NOSCRIPT>

More about scripting

In this article we will show one of the attributes that can be indicated in the SCRIPT label, which indicates the language that we are going to use. In addition, we will show another very useful way of associating Javascript code on the page, by means of an external file. This point is fundamental and we will have to pay special attention, since it is without a doubt the way of working that is used most often.

Interesting methods of placing Javascript scripts

Indicate the language we are using

Updated: Today it is no longer necessary to specify the scripting language we are using. The reason is that Javascript is the only industry accepted language for scripting on web pages. Therefore, the "language" attribute is really unnecessary.

The <SCRIPT> tag has an attribute that is used to indicate the language that we are using, as well as its version. For example, we can indicate that we are programming in Javascript 1.2 or Visual Basic Script, which is another language for programming scripts in the client browser that is only compatible with Internet Explorer.

The attribute in question is "language" and the most common is to simply indicate the language in which the scripts have been programmed. The default language is Javascript, so if we do not use this attribute, the browser will understand that the language with which it is being programmed is Javascript. One detail where people often make mistakes without realizing it is that language is written with two -g- and not with -g- and with -j- as in Spanish.

<script language = "javascript">

Use of the "type" attribute:

When we put a SCRIPT tag we must use the "type" attribute to indicate what type of script encoding we are doing and the language used.

<script type = "text / javascript">

The "type" attribute is necessary for you to correctly validate your document in the most current versions of HTML.

Note on HTML versions: With the advent of HTML5, the need to specify the language and type (language and type attributes) was removed, therefore, with the script tag it is more than enough to open a block of Javascript code. However, in previous versions of HTML we are required to define the "type" attribute. Although in HTML 4.01 transitional we correctly validate the language attribute, it will not validate if we are doing strict HTML, so we do not recommend using "language" in any case. In the examples of DesarrolloWeb.com where language was used, please ignore it.

Include external Javascript files

Another way to include scripts on web pages, implemented as of Javascript 1.1, is to include external files where you can put many functions that are used on the page. Files usually have a .js extension and are included in this way.

<script src = "external_file.js"> </script>

Within the <SCRIPT> tags, any text can be written and will be ignored by the browser, however, browsers that do not understand the SRC attribute will have this text by instructions. It is very rare for this to happen in the browser landscape. So it is not really advisable to put any Javascript code inside a script block that we are using to include an external file.

The file we include (in this case "external_file.js") must contain only Javascript statements. We must not include HTML code of any kind, not even the </SCRIPT> and </SCRIPT> tags.

The Importance of Using External Javascript Code Files

At the didactic level in this manual we use a lot of the practice of

Include the Javascript code inside the HTML document itself. It is something that comes very comfortable to express small Javascript examples. However, at a professional level it is not a good practice, but the recommendation is to put all, or most of your Javascript code in ".js" files that you include externally. There are several reasons to recommend this practice, among which we can highlight:

From the point of view of code separation, each file should have only one language. HTML documents are placed in ".html" files, CSS documents in ".css" files and Javascript also separated in their ".js" files.

The external file allows the browser to cache it, so when the page is revisited, the code is already in the browser and you don't have to redownload it. This point is especially important when we have many pages that load the same Javascript code, since the browser will actually download the ".js" file once and the other times will consume it from the cache, saving transfer and increasing loading speed.

When you have more advanced notions about Javascript you will see that in the script tag you can use attributes like "defer" or "async". Both attributes cause the browser not to stop executing an external script that is in the middle of the HTML content, but to continue analyzing the page and rendering its content while the file is downloaded. Therefore, both attributes optimize the page load and improve the speed with which the browser presents the content to the user.

Note: When you use "defer" the file load is done in parallel with parsing the HTML and other external files, but the execution is deferred until the browser has finished parsing and rendering the page. When you use "async" you mean that the script is downloaded at the same time that the page is analyzed and that, once downloaded, it is executed, although it did not take long to analyze

the page completely.

An example of load and execution of the deferred Javascript we get it like this.

<script src = "deferred_external_file.js" defer> </script>

Given these other interesting uses that exist in Javascript and that we must know in order to take advantage of the possibilities of technology, we must have learned everything essential to start working with Javascript in the context of a web page.

Chapter III

Javascript syntax

We finally start to see Javascript source code! We hope that all the previous information from the Javascript Manual has been assimilated, in which we have basically learned various ways to include scripts on web pages. So far everything we have seen in this manual may have seemed very theoretical, but from now on we hope you find it more enjoyable to start seeing more practical and directly related to programming.

The Javascript language has a syntax very similar to Java because it is based on it. It is also very similar to that of the C language, so if the reader knows any of these two languages it will be easy to handle with the code. Anyway, in the following chapters we are going to describe all the syntax carefully, so newbies will have no problem with it.

Comments in the code

A comment is a part of code that is not interpreted by the browser and whose utility lies in making it easier for the programmer to read. The programmer, as he develops the script, leaves individual phrases or words, called comments, which help him or anyone else to read the script more easily when modifying or debugging it.

Some Javascript comments have been seen previously, but now we are going to count them again. There are two types of comments in the language. One of them, the double bar, is used to comment on a line of code. The other comment can be used to comment on several lines and is indicated with the signs / * to start the comment and * / to end it. Let's see some examples.

```
<SCRIPT>
// This is a one line comment
/ * This comment can be extended
along several lines.
The ones you want * /
</SCRIPT>
```

Upper case and lower case

In Javascript uppercase and lowercase letters must be respected. If we make a mistake when using them, the browser will respond with an error message, either syntax or indefinite reference.

For example, the alert () function is not the same as the Alert () function. The first displays text in a dialog box, and the second (with the first capital A) simply does not exist, unless we define it ourselves. As you can see, for the function to recognize Javascript, you have to write all lowercase. We will see another clear example when we deal with variables, since the names we give to the variables are also case sensitive.

As a general rule, the names of things in Javascript are always written in lowercase, unless a name with more than one word is used, since in this case the initials of the words following the first will be capitalized. For example document.bgColor (which is a place where the background color of the web page is saved), it is written with the capital "C", as it is the first letter of the second word. You can also use capital letters in the initials of the first words in some cases, such as the names of the classes, although we will see later what these cases are and what the classes are.

Separation of instructions

The different instructions that our scripts contain must be conveniently separated so that the browser does not indicate the corresponding syntax errors. Javascript has two ways of separating instructions. The first is through the semicolon (;) character and the second is through a line break.

For this reason Javascript statements do not need to end with a semicolon unless we put two instructions on the same line.

It is not a bad idea, anyway, to get used to using the semicolon after each instruction because other languages such as Java or C force them to be used and we will be getting used to making a syntax more similar to the usual one in advanced programming environments.

Variables in Javascript

This is the first of the articles that we are going to dedicate to variables in Javascript within the Javascript Manual. We will see, if we do not already know, that variables are one of the fundamental elements when making programs, in Javascript as well as in most of the existing programming languages.

So let's start by getting to know the concept of a variable and we'll learn how to declare them in Javascript, along with detailed explanations of their use in the language.

Variable concept

A variable is a space in memory where data is stored, a space where we can store any type of information that we need to carry out the actions of our programs. We can think of it as a box, where we store data. That box has a name, so that later we can refer to the variable, retrieve the data as well as assign a value to the variable whenever we want.

For example, if our program performs sums, it will be very normal for us to store in variables the different addends that participate in the operation and the result of the sum. The effect would be something like this.

adding1 = 23

adding2 = 33

sum = adding1 + adding2

In this example we have three variables, adding1, adding2 and sum, where we save the result. We see that their use for us is as if we had a section where to save a data and that they can be accessed just by putting their name.

Rules for variable naming in Javascript

Variable names must be constructed with alphanumeric characters (numbers and letters), the underscore or underscore (_), and the dollar character $. Apart from this, there are a number of additional rules for constructing names for variables. The most important is that they cannot start with a numeric character. We cannot use rare characters such as the + sign, a space, or a - sign. Supported names for variables could be:

Age

country of birth

_Name

$ element

Other $ _Names

We must also avoid using reserved names as variables, for example we will not be able to call our variable words like return or for, which we will see that they are used for structures of the language itself. Let's now look at some variable names that you are not allowed to use:

12 months

your name

return

for

more or less

pe% pe

Variable names in Javascript are case sensitive

Remember that Javascript is a case-sensitive language, so variables are also affected by that distinction. Therefore, the variable named "myname" is not the same as the variable "myName". "Age" is not the same as "age".

Keep this detail in mind, as it is a common source of code problems that are sometimes difficult to detect. This is because sometimes you think you are using a variable, which should have a certain data, but if you make a mistake when writing it and put upper or lower case where it should not, then it will be another different variable, which will not have the expected data. Since Javascript does not force you to declare variables, the program will run without producing an error, however, the execution will not produce the desired effects.

Declaration of variables in Javascript

Declaring variables consists of defining, and incidentally informing the system, that you are going to use a variable. It is a common custom in programming languages to explicitly specify the variables to be used in programs. In many programming languages there are some strict rules when it comes to declaring variables, but the truth is that Javascript is quite permissive.

Javascript skips many rules for being a somewhat free language when programming and one of the cases in which it gives a little freedom is when declaring the variables, since we are not obliged to do so, contrary to what happens in other programming languages like Java, C, C # and many others.

Variable declaration with var

Javascript has the word "var" that we will use when we want to declare one or more variables. Not surprisingly, that word is used to define the variable before using it.

Note: Although Javascript does not force us to explicitly declare variables, it is advisable to declare them before using them and we will see from now on that it is also a good habit. In addition, in successive articles we will see that in some special cases, a script in which we have declared a variable and another in which we have not, will not produce exactly the same results, since the declaration does not affect the scope of the variables.

var operand1

var operand2

You can also assign a value to the variable when it is being declared

var operand1 = 23

var operand2 = 33

It is also allowed to declare several variables on the same line, provided they are separated by commas.

var operand1, operand2

Declaration of Javascript variables with let and const

From Javascript in modern versions (remember that Javascript is a standard and that as it evolves over time), specifically in Javascript in its ES6 version, there are other ways to declare variables:

Let declaration: This new way of declaring variables affects their scope, since they are local to the block where they are being declared.

Const Declaration: Actually "const" does not declare a variable but a constant, which cannot change its value during the execution of a program.

Perhaps at this point in the Javascript manual it is not necessary to delve too deeply into these declaration models and for those who are learning we recommend focusing on the use of declarations with "var". However, if you want to have more information about these new types of variables, we explain them in the article Let and const: variables in ECMAScript 2015. If you want to see other news about the Javascript standard that came in 2015, we recommend reading the ES6 Manual.

Scope of variables in Javascript

The scope of variables is one of the most important concepts that we must know when working with variables, not only in Javascript, but in most programming languages.

In the previous article we already started to explain what variables are and how to declare them. In this article of the Javascript Manual we intend to explain in detail what this field of variables is and offer examples so that it can be well understood.

At the beginning of the article we will refer to the scope of the variables declared with "var". However there is a newer way to declare variables, with "let", that directly affects your scope. At the end we will also offer some notes and references to understand it.

Variable scope concept

Variables scope is called the place where they are available. In general, when we declare a variable we make it available in the place where it has been declared, this occurs in all programming languages and since Javascript is defined within a web page, the variables that we declare on the page will be accessible within she.

In Javascript we will not be able to access variables that have been defined on another page. Therefore, the page where it is defined is the most common scope of a variable and we will call this type of global variables the page. We will also see that variables can be made with different scopes from the global one, that is, variables that we will declare and will be valid in more limited places.

Global variables

As we have said, global variables are those that are declared in the widest possible scope, which in Javascript is a web page. To declare a global variable to the page we will simply do it in a script, with the word var.

<SCRIPT>

global variable var

</SCRIPT>

Global variables are accessible from anywhere on the page, that is, from the script where they have been declared and all the other scripts on the page, including event handlers, such as onclick, which we already saw could be included within certain HTML tags.

Local variables

We can also declare variables in narrower places, such as a function. We will call these variables local. When local variables are declared, we can only access them within the place where they have been declared, that is, if we had declared them in a function, we can only access them when we are in that function.

Variables can be local to a function, but can also be local to other scopes, such as a loop. In general, any area bounded by keys is a local area.

```
<SCRIPT>
function myFunction () {
   var variableLocal
}
</SCRIPT>
```

In the previous script we have declared a variable within a function, so that variable will only be valid within the function. You can see how the braces are used to delimit the place where that function is defined or its scope.

There is no problem in declaring a local variable with the same name as a global one, in this case the global variable will be visible from the whole page, except in the scope where the local variable is declared since in this site that variable name is busy by the local and it is she who has validity. In summary, the variable that will be valid anywhere on the page is the global one. Less in the area where the local variable is declared, which will be the one that has validity.

```
<SCRIPT>
var number = 2
function myFunction () {
   var number = 19
```

```
   document.write (number) // print 19
}
document.write (number) // print 2
</SCRIPT>
```

Note: To understand this code you will surely find it useful to consult the chapter on creating functions in Javascript.

A tip for beginners might be to not declare variables with the same names, so there is never confusion about which variable is valid at any given time.

Differences between declaring variables with var, or not declaring them

As we have said, in Javascript we are free to declare or not the variables with the word var, but the effects that we will achieve in each case will be different. Specifically, when we use var we are making the varible that we are declaring local to the scope where it is declared. On the other hand, if we do not use the word var to declare a variable, it will be global to the entire page, whatever the scope in which it has been declared.

In the case of a variable declared on the web page, outside of a function or any other smaller scope, it is not important to us whether or not it is declared with var, from a functional point of view. This is because any variable declared outside of a scope is global to the entire page. The difference can be seen in a function for example, since if we use var the variable will be local to the function and if we do not use it, the variable will be global to the page. This difference is fundamental when it comes to correctly controlling the use of variables on the page, since if we do not do it in a function, we could overwrite the value of a variable, losing the data that it may previously contain.

```
<SCRIPT>
var number = 2
function myFunction () {
   number = 19
   document.write (number) // print 19
}
document.write (number) // print 2
// call the function
myFunction ()
document.write (number) // print 19
</SCRIPT>
```

In this example, we have a global variable to the page called number, which contains a 2. We also have a function that uses the variable number without having declared it with var, so the variable number of the function will be the same global variable number declared out of function. In a situation like this, executing the function will overwrite the variable number and the data that was there before executing the function will be lost.

Variable declaration with let

Since ECMAScript 2015 there is the let statement. The syntax is the same as var when declaring variables, but in the case of let the declaration affects the block.

Block means any space delimited by keys, such as the statements inside the keys of a for loop.

```
for (let i = 0; i <3; i ++) {
  // in this case the variable i only exists inside the for loop
  alert (i);
}
// outside the for block there is no variable i
```

If that variable "i" had been declared in the for loop header using "var", it would exist outside the for code block.

What can we save in variables

In a variable we can introduce various types of information. For example we could enter simple text, whole or real numbers, etc. These different kinds of information are known as data types. Each one has different characteristics and uses.

Let's see what are the most common data types of Javascript.

Numbers

To start we have the numeric type, to save numbers like 9 or 23.6

Chains

The character string type stores a text. Whenever we write a character string we must use quotation marks (").

Booleans

We also have the boolean type, which stores information that can be valid whether (true) or not (false).

Finally, it would be relevant to point out here that our variables can contain more complicated things, such as an object, a function, or empty (null), but we will see later.

Actually our variables are not forced to save a specific data type and therefore we do not specify any data type for a variable when we are declaring it. We can introduce any information in a variable of any type, we can even change the content of a variable from one type to another without any problem. We will see this with an example.

```
var city_name = "Valencia"
revised var = true
city_name = 32
revised = "no"
```

This lightness when assigning types to variables can be an advantage at first, especially for inexperienced people, but in the long run it can be a source of errors since depending on the type of variables they will behave in one way or another and If we do not control exactly the type of the variables we can find ourselves adding a text to a number. Javascript will work perfectly, and will return a data, but in some cases it may not be what we were expecting. So, although we have freedom with types, this same freedom makes us be more attentive to possible mismatches that are difficult to detect throughout the programs. Let's see what would happen if we added letters and numbers.

```
var adding1 = 23
var adding2 = "33"
var sum = adding1 + adding2
document.write (sum)
```

This script would show us on the page the text 2333, which does not correspond to the sum of the two numbers, but to their concatenation, one after the other.

Data types in Javascript

In our scripts we are going to handle various kinds of information variables, such as texts or numbers. Each of these kinds of information are the data types. Javascipt distinguishes between three types of data and all the information that can be stored in variables will be embedded in one of these data types. Let's take a closer look at what these three types of data are.

Numeric data type

There is only one numeric data type in this language, unlike in most of the better-known languages. All the numbers are therefore of the numerical type, regardless of the precision they have or if they are real or integer numbers. Integers are non-comma numbers, such as 3 or 339. Real numbers are fractional numbers, such as 2.69 or 0.25, which can also be written in scientific notation, for example 2,482e12.

With Javascript we can also write numbers in other bases, such as hexadecimal. Bases are numbering systems that use more or fewer digits to write numbers. There are three bases with which we can work

Base 10 is the system that we usually use, the decimal system. Any number, by default, is understood to be written in base 10.

Base 8, also called the octal system, which uses digits from 0 to 7. To write a number in octal, just write that number preceded by 0, for example 045.

Base 16 or hexadecimal system, is the numbering system that uses 16 digits, those between 0 and 9 and the letters from A to F, for the missing digits. To write a number in hexadecimal we must write it preceded by a zero and an x, for example 0x3EF.

Boolean type

The type bolean, boolean in English, is used to save a yes or no or in other words, a true or a false. It is used to perform logical operations, generally to perform actions if the content of a variable is true or false.

If a variable is true then ----- I execute some instructions

If not ----- I execute others

The two values that boolean variables can have are true or false.

myBoleana = true

**
**

myBoleana = false

Data type character string

The last type of data is the one used to save a text. Javascript has only one data type to save text and any number of characters can be entered. A text can be composed of numbers, letters and any other type of characters and signs. Texts are written in quotation marks, double or single.

myText = "Pepe is going to fish"

myText = '23 %% $ Letters & * - * '

Everything that is enclosed in quotation marks, as in the previous examples, is treated as a character string regardless of what we put inside the quotation marks. For example, in a text variable we can save numbers and in that case we have to take into account that text and numeric variables are not the same thing and while numerical variables are used to make mathematical calculations, text variables are not .

Escape characters in text strings

There are a series of special characters that are used to express certain controls in a text string, such as a line break or a tab. These are the escape characters and are written with special notation that starts with a backslash (a forward slash opposite the normal one ") and then the code for the character to be displayed is placed.

A very common character is the line break, which is achieved by typing n. Another very common character is to place quotation marks, because if we put quotation marks without their special character, the quotation marks that we put in will start to close the character string. The quotation marks must then be entered with "o '(double or single quotation marks). There are other escape characters, which we will see in the table below, which are more summarized, although the one used to write a backslash should also be highlighted as the usual character. , so as not to confuse it with the start of an escape character, which is the double backslash.

Table with all escape characters

Line break: \ n

Single quote: \ '

Double quote: \ "

Tab: \ t

Carriage return: \ r

Page advance: \ f

Move back space: \ b

**Counterbar: **

Some of these characters will probably never be used by you, as their function is a bit strange and sometimes unclear.

Chapter IV

Javascript operators

When developing programs in any language, operators are used, which are used to make the calculations and operations necessary to carry out your objectives. Even the smallest program imaginable needs operators to do things, since a program that does not perform operations would only limit itself to always doing the same.

It is the result of operations that makes a program vary its behavior according to the data it has to work with and offers us results that are relevant to the user who uses it. There are simpler or more complex operations, which can be performed with operands of different types, such as numbers or texts, we will see in this chapter, and the following, in detail, all these operators available in Javascript.

Examples of using operators

Before going to list the different types of operators, let's see a couple of examples of these to help us get a more accurate idea of what they are. In the first example we are going to perform a sum using the sum operator.

3 + 5

This is a very basic expression that doesn't make much sense on its own. It does the sum between the two operands number 3 and 5, but it does not help much because nothing is done with the result. Usually more than one operator is combined to create more useful expressions. The following expression is a combination between two operators, one performs a mathematical operation and the other is used to save the result.

myVariable = 23 * 5

In the previous example, the operator * is used to perform a multiplication and the operator = is used to assign the result in a variable, so we save the value for later use.

Operators can be classified according to the type of actions they perform. Next, we will look at each of these groups of operators and describe the function of each one.

Arithmetic operators

They are those used to carry out simple mathematical operations such as addition, subtraction or multiplication. In javascript they are the following:

+ Sum of two values

- Subtraction of two values, can also be used to change the sign of a number if we use it with a single operand -23

* Multiplication of two values

/ Division of two values

% The remainder of dividing two numbers (3% 2 would return 1, the remainder of dividing 3 by 2)

++ Increment in one unit, used with a single operand

- Decrement in one unit, used with a single operand

Examples

price = 128 // I introduce a 128 in the price variable

units = 10 // another assignment, then we will see assignment operators

invoice = price * units // multiply price by units, I get the invoice value

remainder = invoice% 3 // I get the remainder from dividing the invoice variable by 3

price ++ // increase the price by one unit (now worth 129)

Assignment operators

They are used to assign values to variables, we have already used the assignment operator = in previous examples, but there are other operators of this type, which come from the C language and that many of the readers will already know.

= Assignment. Assign the right part of the equals to the left part. To the right the final values are placed and to the left a variable is generally placed where we want to save the data.

+ = Assignment with sum. Add the right part to the left part and save the result to the left part.

- = Allocation with subtraction

* = Multiplication allocation

/ = Division assignment

% = The rest is obtained and assigned

Examples

savings = 7000 // assign a 7000 to the savings variable

savings + = 3500 // increase savings variable by 3500, now worth 10500

savings / = 2 // divide my savings by 2, now there are 5250

In the following article we will continue to learn about other Javascript operators: String operators, logical operators and conditional operators.

Chain operators

Character strings, or text variables, also have their own operators to perform typical actions on strings. Although javascript only has one operator for strings, other actions can be performed with a series of predefined functions in the language that we will see later.

+ Concatenate two strings, paste the second string after the first one.

Example

string1 = "hello"

string2 = "world"

concatenated string = string1 + string2 // concatenated string is valid "helloworld"

An important detail that can be seen in this case is that the operator + serves two different uses, if its operands are numbers it adds them, but if it is strings it concatenates them. This happens in general with all the operators that are repeated in the language, javascript is smart enough to understand what type of operation to perform by checking the types that are involved in it.

on with text and numeric operators intermingled. In this case javascript assumes that you want to concatenate and treats the two operands as if they were character strings, even if the text string we have was a number. We will see this more easily with the following example.

myNumber = 23

myString1 = "pepe"

myString2 = "456"

result1 = myNumber + myString1 // result1 is worth "23pepe"

result2 = myNumber + myString2 // result2 is valid "23456"

myString2 + = myNumber // myString2 is now "45623"

As we have seen, also in the case of the operator + =, if we are dealing with text strings and intermixed numbers, it will treat the two operators as if they were strings.

Note: As you may have imagined, many typical operations to perform with strings are missing, for which there are no operators. It is because these functionalities are obtained through the String class of Javascript, which we will see later.

Logical operators

These operators are used to perform logical operations, which are those that result in a true or a false, and are used to make decisions in our scripts. Instead of working with numbers, to perform this type of operations, Boolean operands, which we knew earlier, are used, which are true (true) and false (false). Logical operators relate Boolean operands to result in another Boolean operand, as we can see in the following example.

If I am hungry and I have food then I start eating

Our Javascript program would use a Boolean operand in this example to make a decision. First it will see if I am hungry, if it is true (true) it will see if I have food. If both are true, it can be eaten. In case I don't have food or I'm not hungry I wouldn't eat, just like if I'm not hungry or food. The operand in question is the AND operand, which will be true if the two operands are true.

Note: In order not to be misleading, it should be said that logical operators can be used in combination with data types other than Boolean, but in this case we must use them in expressions that make them Boolean. In the next group of operators that we are going to deal with in this article we will talk about conditional operators, which can be used together with logical operators to make all the complex sentences that we need. For example:

```
if (x == 2 && y! = 3) {
  // the variable x is worth 2 and the variable y is different from three
}
```

In the above conditional expression we are evaluating two checks that relate to a logical operator. On the one hand x == 2 will return true if the variable x is 2, and on the other, y! = 3 will return true when the variable y has a value other than 3. Both checks return one boolean each, which then The logical

operator && is applied to it to check if both checks were fulfilled at the same time.

It goes without saying that, to see examples of conditional operators, we need to learn control structures like if, which we have not yet reached.

! Operator NO or denial. If it was true, it changes to false and vice versa.

&& Operator And, if they are both true, it is true.

|| Operator O, true if at least one of them is true.

Example

myBoleano = true

myBoleano =! myBoleano // miBoleano now worth false

hunger = true

gotFood = true

comoMeal = I am hungry && I have food

Conditional operators

They are used to make conditional expressions as complex as we want. These expressions are used to make decisions based on the comparison of various elements, for example if one number is greater than another or if they are the same.

Note: Conditional operators are of course also used to make expressions comparing other data types. Nothing prevents comparing two chains, to see if they are the same or different, for example. We could even compare Booleans.

Conditional operators are used in conditional expressions for decision making. As these conditional expressions will be studied later it will be better to describe the conditional operators later. Anyway here we can see the table of conditional operators.

== Check if two values are equal

! = Check if two values are different

> Greater than, returns true if the first operand is greater than the second

<Less than, is true when the item on the left is less than the one on the right

> = Greater equal

<= Less equal

We will see examples of conditional operators when we explain control structures, such as the if conditional.

Additionally, in this text we will see an issue of considerable importance in programming in general, which is the precedence of operators, which we must take into account whenever we use different operators in the same expression, so that they are related to each other and are resolved the way we had planned.

Bit-level operators

These are very unusual and you may never get to use them. Its use is made to carry out operations with zeros and ones. All that a computer handles are zeros and ones, although we

The numbers and letters for our variables actually these values are written internally in the form of zeros and ones. In some cases we may need to perform operations treating the variables as zeros and ones and for this we will use these operands. In this manual it is a bit too big for us to carry out a discussion about this type of operators, but here you can see these operators in case you ever need them.

& Y bit

^ Bit Xor

| Or bit

<< >> >>> >>> = >> = << = **Various kinds of changes**

Operators precedence

The evaluation of a sentence that we have seen in the previous examples is quite simple and easy to interpret, but when a multitude of different operators come into play in a sentence, there may be confusion when interpreting it and elucidating which operators are the ones that they run before others. To set some guidelines in the evaluation of the sentences and that they are always executed the same and with common sense, there is the precedence of operators, which is nothing more than the order in which the operations they represent will be executed. Initially, all operators are evaluated from left to right, but there are additional rules, whereby certain operators are evaluated before others. Many of these rules of precedence are derived from mathematics and are common to other languages, we can see them below.

() []. Parentheses, square brackets, and the dot operator for objects

! - ++ - negation, negative and increments

*** /% Multiplication division and module**

+ - Add and subtract

<< >> >>> Bit level changes

<<=>> = Conditional operators

==! = Conditional operators of equality and inequality

& ^ | Bit-level logic

&& || Boolean logicians

= + = - = * = / =% = << = >> = >>> = & = ^ =! = Assignment

In the following examples we can see how expressions could become confusing, but with the operator precedence table we will be able to understand without errors what is the order in which they are executed.

12 * 3 + 4 - 8/2% 3

In this case, the * / and% operators are executed first, from left to right, with which these operations would be performed. First the multiplication and then the division because it is more to the left of the module.

36 + 4 - 4% 3

Now the module.

36 + 4 - 1

Finally the additions and subtractions from left to right.

40 - 1

Which gives us the following value.

39

In any case, it is important to realize that the use of parentheses can save us a lot of headaches and, above all, the need to know the operator precedence table by heart. When we see unclear the order in which the sentences will be executed, we can use them and thus force that the piece of expression found inside the parentheses be evaluated before.

Javascript typeof operator for type control

We have been able to verify that, for certain operators, the type of data they are handling is important, since if the data is of one type, different operations will be performed than if they are of another.

For example, when we used the operator +, if it was numbers it added them, but if it was character strings it concatenated them. We see then that the type of data we are using does matter and that we will have to be aware of this detail if we want our operations to be carried out as expected.

To check the type of a data you can use another operator that is available as of javascript 1.1, the typeof operator, which returns a text string that describes the type of the operator that we are checking.

Note: throughout our experience with Javascript we will see that many times it is more useful to change the data type of a variable before doing a check with tipeof to see if we can use it as an operand. There are several functions to try to change the type of a variable, such as parseInt (), which we will see later in the Second Part of the Javascript Manual.

```
boolean var = true
numeric var = 22
floating_number = 13.56
var text = "my text"
var date = new Date ()
document.write ("<br> Boolean type is:" + boolean typeof)
document.write ("<br> The numeric type is:" + numeric typeof)
document.write ("<br> The type of floating_number is:" + typeof floating_number)
document.write ("<br> The text type is:" + typeof text)
document.write ("<br> The date type is:" + typeof date)
```

If we execute this script we will obtain that the following text will be written on the page:

Boolean type is: boolean

The numeric type is: number

The type of floating_numeric is: number

The text type is: string

The date type is: object

In this example we can see that the different types of variables are printed on the page. These can be the following:

boolean, for boolean data. (True or false)

number, for the numerical ones.

string, for character strings.

object, for objects.

function, for functions.

undefined, for declared variables that have not been assigned values.

We want to highlight just two more details:

Numbers, whether or not they have a decimal part, are always of the numeric data type.

One of the variables is a little more complex, it is the date variable which is an object of the Date () class, which is used for handling dates in scripts. We will see it later, as well as the objects.

Also note that Javascript functions are treated as the

Recommendations with typeof

In our opinion it is not very appropriate to constantly ask the type of a variable, to do different things when it has one type or another. The normal thing is that you are clear about what type of operation you want to perform at a given moment and that, if you are not sure if the variable of one type or another has arrived, you convert it with some function such as parseInt () or parseFloat (), that we will see later.

The most useful case I can now remember is asking if the type of a variable is undefined, which can give you the answer about whether the variable has been initialized or not.

```
variable let;
if (typeof variable == 'undefined') {
console.log ('The variable is undefined, it has no defined value);
}
```

Another case that is often used a lot is when debugging, when you need to find errors in your code, in the face of erratic and unexpected operations. In those cases it is useful to consult the type of the variables, because sometimes in Javascript they bring you surprises that make your programs not work well.

Javascript control structures

The scripts seen so far in the Javascript Manual have been tremendously simple and linear: simple statements were executed one after the other from start to finish. However, this does not always have to be the case and in fact, in most cases things are much more complex.

If we have any experience in programming we will know that in programs it will generally be necessary to do different things depending on the state of our variables or to carry out the same process many times without writing the same lines of code over and over again. Control structures are used to do more complex things in our scripts. With them we can make decisions and loops. In the following chapters we will learn about the different control structures that exist in Javascript.

Decision making

They are used to perform some actions or others depending on the state of the variables. That is, make decisions to execute some instructions or others depending on what is happening at that moment in our programs.

For example, depending on whether the user who enters our page is of legal age or not, we may or may not allow him to view the contents of our page.

If age is greater than 18 then

I let you see the adult content

If not

I'm sending you off the page

In Javascript we can make decisions using two different statements.

IF

SWITCH

Loops

Loops are used to perform certain actions repeatedly. They are widely used at all levels in programming. With a loop we can, for example, print the numbers from 1 to 100 on a page without having to write the print instruction a hundred times.

From 1 to 100

Print the current number

In javascript there are several types of loops, each one is indicated for a different type of iteration and they are the following:

FOR

WHILE

DO WHILE

As we have already pointed out, control structures are very important in Javascript and in any programming language. That is why in the following chapters we will see each of these structures carefully, describing their use and offering some examples.

IF structure in Javascript

In the Javascript Manual of DesarrolloWeb.com we have already begun to explain what control structures are. In this article, we are going to show how the if statement works, which is the most common structure used to make decisions in computer programs.

IF is a control structure used to make decisions. It is a conditional that is used to perform one or the other operations based on an expression. It works as follows, an expression is evaluated first, if it gives a positive result the actions related to the positive case are performed.

The syntax of the IF structure is as follows.

Note: All control structures are lowercase in Javascript. Although sometimes we can write the name of the structure in the text of the manual in capital letters, in the code of our scripts we always have to put it in lower case. Otherwise we will receive an error message.

```
if (expression) {
  // actions to be taken in the positive case
  // ...
}
```

Optionally, actions can be indicated in case the evaluation of the sentence returns negative results.

```
if (expression) {
  // actions to be taken in the positive case
  // ...
```

```
} else {
  // actions to be taken in the negative case
  // ...
}
```

Let's look at several things. To start, we see how the keys include the actions that we want to carry out in case the expressions are fulfilled or not. These keys must always be placed, except in the case that there is only one instruction as actions to be performed, which are optional.

Note: Although the keys to encompass the statements to be executed in both the positive and negative cases are optional when we want to execute a single statement, the recommendation is to always place them, because we will thus obtain a clearer source code. For example:

```
if (it rains)
  alert ("Water falls");
```

It would be exactly like this code:

```
if (it rains) {
  alert ("Water falls");
}
```

Or even, just like this other:

```
if (it rains) alert ("Water falls");
```

However, when we use the keys, the code is much clearer, because you can see at a glance what instructions are depending on the positive case of the if. This is a detail that may not be very important now, but it will be appreciated when the program is more complex or when several programmers are in charge of touching the same code.

Another detail that is obvious is the indentation (margin) that we have placed in each of the instruction blocks to be executed in the positive and negative cases. This indentation is totally optional, we have only done it this way so that the IF structure is understood in a more visual way. Line breaks are also not necessary and have also been placed for a better view of the structure. We could perfectly put the entire IF statement in the same line of code, but that won't help things be clear.

Note: We, as well as anyone with some experience in the programming area, would advise using the necessary indents and line breaks so that the instructions can be better understood. Maybe the day you make a code you will be clear about what you have done and why it is so, but in a month, when you have to reread that code, you may remember less than what you did in your scripts and you will appreciate that they have a friendly format for that can be easily read by people. If you work as a team these recommendations will be even more important, since it is even more difficult to read source code that other people have made.

Let's see some example of IF conditionals.

```
if (dia == "monday")
   document.write ("Have a happy start to the week")
```

If it is Monday you will wish us a happy week. It will not do otherwise. As in this example we only indicate an instruction for the positive case, it will not be necessary to use the keys (although it would be advisable to have put them). Also note the conditional operator consisting of two equal signs.

Let's now see another example, a little longer.

```
if (credit> = price) {
  document.write ("you have bought the article" + new article) // show purchase
  cart + = newArticle // insert the item in the shopping cart
```

```
  credit - = price // I decrease the credit according to the price of the item
} else {
  document.write ("you have run out of credit") // I report that you are short of money
  window.location = "shoppingcart.html" // I go to the cart page
}
```

This example is a little more complex, and also a little fictional. What I do is check if I have credit to make an alleged purchase. To do this, I look at whether the credit is greater than or equal to the price of the item, if so, I inform about the purchase, insert the item in the cart and rest the price of the accumulated credit. If the price of the item is higher than the money available, I report the situation and send the browser to the page where your shopping cart is displayed.

Conditional expressions

The expression to evaluate is always placed in parentheses and is made up of variables that are combined with each other using conditional operators. We recall that conditional operators related two variables and always returned a boolean result. For example, a conditional operator is the operator "is equal" (==), which returns true if the two operands are the same or false if they are different.

if (age> 18)

document.write ("you can see this page for adults")

In this example we use the conditional operator "is greater" (>). In this case, it returns true if the variable age is greater than 18, so the following line would be executed, informing us that the adult content can be seen.

Conditional expressions can be combined with logical expressions to create more complex expressions. We remember that logical expressions are those that have Booleans as operands and that return another Boolean value. They are the logical negation, logical AND, and logical OR operators.

if (battery <0.5 && redElectrica == 0)

document.write ("your laptop will shutdown in seconds")

What we do is check if the battery of our supposed computer is less than 0.5 (it is almost finished) and we also check if the computer has no electrical network (it is unplugged). Then the logical operator matches them to a Y, so if it is almost without battery AND without power, it reports that the computer is going to shut down.

Note: The list of operators that can be used with IF structures can be seen in the chapter on conditional operators and logical operators.

The if structure is one of the most used in programming languages, to make

decisions based on the evaluation of a sentence. In the previous article in the Javascript Manual we already started to explain the if structure and now we are going to see some more advanced uses.

Nested IF statements

To make conditional structures more complex we can nest IF statements, that is, put IF structures inside other IF structures. With a single IF we can evaluate and perform one action or another according to two possibilities, but if we have more possibilities to evaluate we must nest IFs to create the necessary code flow to decide correctly.

For example, if I want to check if one number is less than or equal to another, I have to evaluate three different possibilities. First I can check if the two numbers are equal, if they are, I have already solved the problem, but if they are not equal I will still have to see which of the two is greater. Let's look at this example in Javascript code.

```
var number1 = 23
var number2 = 63
if (number1 == number2) {
  document.write ("The two numbers are the same")
} else {
  if (number1> number2) {
    document.write ("The first number is greater than the second")
  } else {
    document.write ("The first number is less than the second")
  }
}
```

The flow of the program is as we mentioned before, first it is evaluated if the two numbers are equal. If positive, a message is displayed informing you of this. Otherwise we already know that they are different, but we still have to find out which of the two is greater. For this, another comparison is made to find out if the first is greater than the second. If this comparison gives

positive results, we display a message saying that the first is greater than the second, otherwise we will indicate that the first is less than the second.

We note again that the keys are optional in this case, since only one statement is executed for each case. In addition, line breaks and indents are also optional in any case and only serve to see the code in a more orderly way. Keeping the code well structured and written in an understandable way is very important, since it will make life more pleasant when programming and later when we have to review the programs.

Note: In this manual I will use a notation like the one you have seen in the previous lines. Also, I will keep that notation at all times. This will undoubtedly make the codes with examples easier to understand, if we did not do so it would be a real nuisance to read them. This same recipe is applicable to the codes that you have to create and the main beneficiary will be yourself and the colleagues who get to read your code.

IF operator

There is an operator that we haven't seen yet and it's a more schematic way of doing some simple IFs. It comes from the C language, where very few lines of code are written and where the less we write the more elegant we will be. This operator is a clear example of saving lines and characters when writing scripts. We will see it quickly, because the only reason I include it is so that you know that it exists and if you find it on some occasion out there you can identify it and how it works.

An example of using the IF operator can be seen below.

Variable = (condition)? value1: value2

This example not only performs a value comparison, it also assigns a value to a variable. What it does is evaluate the condition (placed in parentheses) and if it is positive, it assigns the value1 to the variable and otherwise it assigns it the value2. Let's see an example:

moment = (current_time <12)? "Before noon": "After noon"

This example looks to see if the current time is less than 12. If so, it is now before noon, so assign "Before Noon" to the variable time. If the hour is greater than or equal to 12, it is already after noon, so the text "After noon" is assigned to the moment variable.

Javascript SWITCH structure

Control structures are the way in which you can control the flow of programs, to do different things depending on the states of the variables. In the Javascript Manual we already started to see the control structures and now it is SWITCH's turn, a slightly more complex structure that allows multiple operations depending on the state of a variable.

In this article we will see that switch helps us to make decisions based on different states of the variables. This expression is used when we have multiple possibilities as a result of evaluating a sentence.

The SWITCH structure was incorporated from version 1.2 of Javascript (Netscape 4 and Internet Explorer 4). Its syntax is as follows.

```
switch (expression) {
  case value1:
    Statements to execute if the expression has a value of value1
    break
  case value2:
    Statements to execute if the expression has a value of value2
    break
  case value3:
    Statements to execute if the expression is valued at value3
    break
  default:
    Statements to execute if the value is not one of the above
}
```

The expression is evaluated, if it is worth value1 the sentences related to that case are executed. If the expression is worth value2, the instructions related to that value are executed, and so on, for as many options as we want. Finally, for all cases not previously contemplated, the default case is executed.

The word break is optional, but if we do not put it after a value is found, all the sentences related to this and all the following will be executed. That is, if in our previous scheme there were no breaks and the expression were value1, the sentences related to value1 would be executed, as well as those related to value2, value3 and default.

The default option or default option is also optional.

Let's see an example of using this structure. Suppose we want to indicate what day of the week it is. If the day is 1 (Monday) get a message indicating it, if the day is 2 (Tuesday) we must get a different message and so on for each day of the week, except on the 6 (Saturday) and 7 (Sunday) that we want show the message "it's weekend". For days greater than 7 we will indicate that this day does not exist.

```
switch (day_of_the_week) {
  case 1:
    document.write ("It's Monday")
    break
  case 2:
    document.write ("It's Tuesday")
    break
  case 3:
    document.write ("It's Wednesday")
    break
```

```
  case 4:
    document.write ("It's Thursday")
    break
  case 5:
    document.write ("It's Friday")
    break
  case 6:
  case 7:
    document.write ("It's weekend")
    break
  default:
    document.write ("That day does not exist")
}
```

The example is relatively simple, it can only have a small difficulty, consisting of interpreting what happens in case 6 and 7, which we had said that we had to show the same message. In case 6, we do not actually indicate any instructions, but since we do not place a break, the following sentence or sentences will be executed, which correspond to the sentence indicated in case 7, which is the message that informs that it is weekend. If the case is 7, it is simply indicated that it is a weekend, as intended.

Note: We also have a video tutorial on SWITCH in Javascript that can be very helpful to understand everything much better.

FOR loop in Javascript

The FOR loop is used to repeat one or more instructions a certain number of times. Among all the loops, the FOR is usually used when we know for sure the number of times we want it to run. The for loop syntax is shown below.

```
for (initialization; condition; update) {
  // statements to execute in each iteration
}
```

The FOR loop has three parts included in parentheses, which help us define how we want repetitions to be performed. The first part is initialization, which is executed only when starting the first iteration of the loop. In this part, the variable that we will use to keep count of the times the loop is executed is usually placed.

The second part is the condition, which will be evaluated every time an iteration of the loop begins. Contains an expression to decide when to stop the loop, or rather, the condition that must be met for the loop to continue running.

Finally we have the update, which serves to indicate the changes that we want to execute in the variables every time the loop iteration ends, before checking if it should continue executing.

After the for, the sentences that we want to execute in each iteration are placed, enclosed in curly braces.

An example of using this loop can be seen below, where the numbers from 0 to 10 will be printed.

```
var i
for (i = 0; i <= 10; i ++) {
  document.write (i)
  document.write ("<br>")
}
```

In this case, the variable i is initialized to 0. As a condition to perform an iteration, it must be met that the variable i is less than or equal to 10. As an update, the variable i will be increased by 1.

As you can see, this loop is very powerful, since in a single line we can indicate many different and very varied things, which allows a fast configuration of the loop and enormous versatility.

For example, if we want to write the numbers from 1 to 1,000 two by two, the following loop will be written.

```
for (i = 1; i <= 1000; i + = 2)
  document.write (i)
```

If we look, in each iteration we update the value of i increasing it by 2 units.

Note: Another detail, we do not use the keys encompassing the instructions of the FOR loop because it only has one statement and in this case it is not forced, as it happened with the IF instructions.

If we want to count down from 343 to 10 we would use this loop.

```
for (i = 343; i> = 10; i--)
  document.write (i)
```

In this case, we decrease the variable i by one unit in each iteration, starting at the value 343 and as long as the variable has a value greater than or equal to 10.

Sample for-loop exercise

We are going to pause to assimilate the for loop with an exercise that does not pose any difficulties if we have understood the operation of the loop.

This is a loop that writes the headings from <H1> to <H6> on a web page with a text that says "Level x heading".

What we want to write to a web page using Javascript is the following:

<H1> Level 1 Header </H1>

<H2> Level 2 Header </H2>

<H3> Level 3 Header </H3>

<H4> Level 4 Header </H4>

<H5> Level 5 Header </H5>

<H6> Level 6 Header </H6>

For this we have to make a loop that starts at 1 and ends at 6 and in each iteration we will write the heading that it touches.

```
for (i = 1; i <= 6; i ++) {
  document.write ("<H" + i + "> Level header" + i + "</ H" + i + ">")
}
```

Now that we are familiar with the for loop, we are in a position to learn how to handle other control structures for repeating, such as the while and do ... while loops.

WHILE and DO WHILE loops

We are dealing with the different control structures that exist in the Javascript language and specifically looking at the different types of loops that we can implement in this programming language. In previous articles of the Javascript Manual we already saw the first of the loops that we should know, the for loop and now we are going to deal with the other two types of control structures to do repetitions. So let's now look at the two types of WHILE loops that we can use in Javascript and the uses of each.

WHILE loop

These loops are used when we want to repeat the execution of some sentences an indefinite number of times, as long as a condition is met. It is easier to understand that the FOR loop, since it does not incorporate in the same line the initialization of the variables its condition to continue executing and its update. Only the condition that must be met for an iteration to be performed is indicated, as we will see below.

```
while (condition) {
  // statements to execute
}
```

An example of code where this loop is used can be seen below.

```
var color = ""
while (color! = "red") {
  color = prompt ("give me a color (write red to exit)", "")
}
```

This is an example of the simplest thing to do with a while loop. What it does is ask the user to enter a color and it does it repeatedly, as long as the entered color is not red. To execute a loop like this we first have to initialize the variable that we are going to use in the loop iteration condition. With the initialized variable we can write the loop, which will check to execute that the color variable is different from "red". In each iteration of the loop a new color is requested from the user to update the color variable and the iteration is finished, so we return to the beginning of the loop, where we have to re-evaluate if what is in the color variable is "red" and so on as long as the text "red" has not been entered as a color.

Note: In this example we have used the Javascript prompt function, which we have not yet seen in this manual. This function is used to display a dialog box where the user must type a text. This function belongs to the Javascript window object and we discuss it in the article Window methods in Javascript.

DO ... WHILE loop

The do ... while loop is the last of the structures to implement repetitions available in Javascript and is a variation of the while loop seen above. It is generally used when we do not know how many times the loop will be executed, just like the WHILE loop, with the difference that we know for sure that the loop will execute at least once.

This type of loop was introduced in Javascript 1.2, so not all browsers support it, only version 4 or higher. In any case, any code you want to write with DO ... WHILE can also be written using a WHILE loop, so in older browsers you will have to translate your DO ... WHILE loop into a WHILE loop.

The syntax is as follows:

```
do {
  // loop statements
} while (condition)
```

The loop is always executed once and at the end the condition is evaluated to tell if the loop is executed again or its execution ends.

Let's look at the example we wrote for a WHILE loop in this other type of loop.

```
var color
do {
  color = prompt ("give me a color (write red to exit)", "")
```

} while (color! = "red")

This example works exactly the same as the previous one, except that we did not have to initialize the color variable before entering the loop. Asks for a color while the entered color is different from "red".

Example of the use of while loops

Let's see below a more practical example on how to work with a WHILE loop. As it is very difficult to make practical examples with the little we know about Javascript, we are going to advance an instruction that we do not yet know.

In this example we are going to declare a variable and initialize it to 0. Then we will add a random number from 1 to 100 to that variable until we add 1,000 or more, printing the value of the sum variable after each operation. It will be necessary to use the WHILE loop because we do not know exactly the number of iterations that we will have to carry out (it will depend on the random values that are obtained).

```
var sum = 0
while (sum <1000) {
  sum + = parseInt (Math.random () * 100)
  document.write (sum + "<br>")
}
```

We assume that as far as the WHILE loop is concerned there will be no problems, but where there may be problems is in the statement used to take a random number. However, it is not necessary to explain the sentence here because we plan to do it later. Anyway, if you want, you can see this article that talks about random numbers in Javascript.

Break and continue

Javascript has different control structures to implement loops, such as FOR, WHILE and DO ... WHILE, which we have already been able to explain in previous chapters of the Javascript Manual. As we have seen, with these loops we can cover a large number of needs, but perhaps over time you will find that you are missing some possibilities of controlling the repetitions of the loops.

Imagine for example that you are doing a very long loop to find something on hundreds or thousands of sites. But put yourself in the case that during the first iterations you find that value you were looking for. Then there would be no point in going through the rest of the loop to search for that item, as you had already found it. In these situations it is convenient for us to know for the loop to cancel the rest of the iterations. Obviously, this is just one example of how we might need to control the loop a bit more. In real life as a programmer you will find many other occasions where you will be interested in doing this or other things with them.

Thus, there are two instructions that can be used in the different control structures and mainly in the loops, which will help you control two types of situations. They are the break and continue instructions:

break: It means stopping the execution of a loop and leaving it.

continue: Used to stop the current iteration and return to the beginning of the loop to perform another iteration, if applicable.

Break

A loop is stopped using the word break. Stopping a loop means exiting it and leaving everything as is to continue the program flow immediately after the loop.

```
for (i = 0; i <10; i ++) {
  document.write (i)
  type = prompt ("tell me if I keep asking ...", "yes")
  if (type == "no")
    break
}
```

This example writes the numbers from 0 to 9 and in each iteration of the loop asks the user if they want to continue. If the user says anything, it continues, except when he says "no", a situation in which he exits the loop and leaves the account where he had left off.

Continue

It is used to return to the beginning of the loop at any time, without executing the lines below the word continue.

```
var i = 0
while (i <7) {
  increase = prompt ("The account is at" + i + ", tell me if I increase",
"if")
  if (increment == "no")
    continue
  i ++
}
```

This example would normally count from i = 0 to i = 7, but each time the loop is executed it asks the user if he wants to increase the variable or not. If you enter "no", the continue statement is executed, which returns to the beginning of the loop without increasing the variable i by 1, since the

statements below the continue would be ignored.

Additional example of the break statement

A more practical example of these instructions can be seen below. It is a FOR loop planned to go up to 1,000 but we are going to stop it with a break when we reach 333.

```
for (i = 0; i <= 1000; i ++) {
  document.write (i + "<br>")
  if (i == 333)
    break;
}
```

Nested loops in Javascript

In the Javascript Manual we have already covered several articles to talk about loops. At this time there should be no problem to create the different types of loops without problems, however, we want to dedicate an entire article to discuss one of the most common uses of loops, which we can find when we are making more complex programs : nesting loops.

Nesting a loop consists of putting that loop inside another. Loop nesting is necessary to make certain processing a little more complex than what we have seen in the previous examples. If in your experience as programmers you have nested them a loop yet, be sure that sooner or later you will meet that need.

A nested loop has a structure like the one below. We will try to explain it in view of these lines:

```
for (i = 0; i <10; i ++) {
  for (j = 0; j <10; j ++) {
  document.write (i + "-" + j)
  }
}
```

Execution will work as follows. To start, the first loop is initialized, so variable i will be 0, and then the second loop is initialized, so variable j will also be 0. In each iteration, the value of variable i is printed, a hyphen (" - ") and the value of the variable j, since the two variables are worth 0, the text" 0-0 "will be printed on the web page.

Due to the flow of the program in nesting schemes like the one we have seen, the loop that is nested (further in) is the one that is executed the most times. In this example, for each iteration of the outermost loop the nested loop will

run full once, that is, it will do its 10 iterations. These values would be written to the web page, in the first iteration of the external loop and from the beginning:

0-0
0-1
0-2
0-3
0-4
0-5
0-6
0-7
0-8
0-9

For each iteration of the outer loop the 10 iterations of the inner or nested loop will be executed. We have seen the first iteration, now we are going to see the following iterations of the outer loop. In each one accumulates a unit in variable i, with what these values would come out.

1-0
1-1
1-2
1-3
1-4
1-5
1-6
1-7

1-8

1-9

And then these:

2-0

2-1

2-2

2-3

2-4

2-5

2-6

2-7

2-8

2-9

So until the two loops are finished, which would be when the value 9-9 is reached.

Let's see an example very similar to the previous one, although a little more useful. The aim is to print all the multiplication tables on the page. From 1 to 9, that is, the table from 1, the table from 2, from 3 ...

```
for (i = 1; i <10; i ++) {
  document.write ("<br> <b> The" + i + "table: </b> <br>")
  for (j = 1; j <10; j ++) {
    document.write (i + "x" + j + ":")
    document.write (i * j)
```

```
    document.write ("<br>")
  }
}
```

With the first loop we control the current table and with the second loop we develop it. In the first loop we write a header, in bold, indicating the table we are writing, first that of 1 and then the others in ascending order up to 9. With the second loop I write each of the values in each table. You can see the example in progress at this link.

Note: We will see more things with nested loops in later chapters, although if we want to go a little ahead to see a new example that strengthens this knowledge, we can see an example in the Javascript Workshop on nested loops, where the table is built with all the colors cigars in 256 color definitions.

Javascript functions

We continue working and expanding our knowledge of Javascript. With what we have seen so far in the Javascript Manual we already have a certain fluency to work in this interesting programming language. But we still have a long way to go.

Now we are going to see a very important topic, especially for those who have never programmed and with Javascript they are taking their first steps in the world of programming since we will see a new concept, that of function, and the uses it has. For those who already know the concept of function it will also be a useful chapter, because we will also see the syntax and operation of functions in Javascript.

What is a function

When making a slightly large program, there are certain processes that can be conceived independently, and which are easier to solve than the whole problem. Furthermore, these are usually carried out repeatedly throughout the execution of the program. These processes can be grouped into a function, defined so that we do not have to repeat that code over and over in our scripts, but simply call the function and it is in charge of doing everything it should.

So we can see a function as a series of instructions that we include within the same process. This process can then be executed from any other site just by calling it. For example, on a web page there may be a function to change the color of the background and from any point on the page we could call it to change the color when we want.

Note: If we want, we can expand this description of the functions in the article Function concept.

The functions are constantly used, not only those that you write, but also those that are already defined in the system, since all programming languages usually have a lot of functions to carry out common processes, such as obtaining the time, printing a message on the screen or convert variables from one type to another. We've already seen some function in our simple examples above. For example, when we were making a document.write () we were actually calling the write () function associated with the page document, which writes text to the page.

In the chapters of functions we will first see how to perform our own functions and how to call them later. Throughout the manual we will see many of the functions defined in Javascript that we must use to perform different types of common actions.

How to write a function

A function must be defined with a special syntax that we will know next.

```
function functionname () {
  function instructions
  ...
}
```

First write the word function, reserved for this use. Then the name of the function is written, which, like variable names, can have numbers, letters and some additional character as in underscore. Then the different instructions of the function are placed in braces. The braces in the case of functions are not optional, and it is also useful to always place them as seen in the example, so that the structure of instructions included in the function is easily recognized.

Let's see an example of a function to write a welcome message on the page inside <H1> tags so that it is more highlighted.

```
function write Welcome () {
  document.write ("<H1> Hello everyone </H1>")
}
```

Just write text on the page. We admit that it is such a simple function that the example does not sufficiently express the concept of a function, but we will see other more complex ones. H1 tags are not shown on the page, but are interpreted as the meaning of the page, in this case we write a level 1 header. Since we are writing on a web page, when putting HTML tags they are interpreted as what they are .

How to call a function

To execute a function we have to invoke it anywhere on the page. With this we will get all the instructions that the function has between the two keys to be executed.

To execute the function we use its name followed by the parentheses. For example, this would call the writeWelcome () function we just created.

write Welcome ()

Where do we put the Javascript functions

Functions are one of the main components of programs, in most programming languages. In the Javascript Manual we have already begun to explain what a function is and how we can create and invoke it in this language. Now we are going to deal with a topic that is not so much about syntax and programming, but has more to do with the correct and habitual use that is made of the functions in Javascript, which is none other than the placement of the code of the functions in the Web page.

In principle, we can place the functions anywhere on the page, always between <SCRIPT> tags, of course. However, there is a limitation when placing it in relation to the places from which it is called. We anticipate that the easiest is to place the function before any call to it and thus we will surely never make a mistake.

There are two possible options to put the code of a function:

a) Put the function in the same script block: Specifically, the function can be defined in the <SCRIPT> block where the function call is, although it does not matter if the call is before or after the function code , within the same <SCRIPT> block.

```
<SCRIPT>
myFunction ()
function myFunction () {
  // I do something ...
  document.write ("This is fine")
}
</SCRIPT>
```

This example works correctly because the function is declared in the same block as its call.

b) Put the function in another script block: It is also valid that the function is in a <SCRIPT> block before the block where the call is.

```
<HTML>
<HEAD>
  <TITLE> MY PAGE </TITLE>
<SCRIPT>
function myFunction () {
  // I do something ...
  document.write ("This is fine")
}
</SCRIPT>
</HEAD>
<BODY>

<SCRIPT>
myFunction ()
</SCRIPT>

</BODY>
</HTML>
```

We see a complete code on how a web page could be where we have Javascript functions. As you can see, the functions are at the top of the page (inside the HEAD). This is an excellent place to put them, because it is

assumed that they will not be used in the header yet and we can always enjoy them in the body because we know for sure that they have already been declared.

To make this issue of function placement clear, let's look at the following example, which would give an error. Look carefully at the following code, which will throw an error, because we make a call to a function that is declared in a subsequent <SCRIPT> block.

```
<SCRIPT>
myFunction ()
</SCRIPT>

<SCRIPT>
function myFunction () {
  // I do something ...
  document.write ("This is fine")
}
</SCRIPT>
```

With this we hope to have resolved all the doubts about the placement of the code of the Javascript functions. In the following articles we will see other interesting topics such as the parameters of the functions.

Function parameters

In the Javascript Manual we have previously talked about functions. Specifically, this is the third article that we address on the subject.

The ideas that we have previously explained about functions are not the only ones that we must learn to handle them to their full potential. Functions also have data input and output. In this article we will see how we can send data to Javascript functions.

Parameters

Parameters are used to send values to functions. A function will work with the parameters to perform the actions. To put it another way, parameters are the input values that a function receives.

To give an example easy to understand, a function that performs a sum of two numbers would have those two numbers as parameters. The two numbers are the input, as well as the output would be the result of the sum, but we will see that later.

Let's look at a previous example in which we created a function to display a welcome message on the web page, but now we are going to pass a parameter that will contain the name of the person to be greeted.

```
function write Welcome (name) {
  document.write ("<H1> Hello" + name + "</H1>")
}
```

As we can see in the example, to define a parameter in the function we have to put the name of the variable that is going to store the data that we pass to it. That variable, which in this case is called name, will have as value the data that we pass to the function when we call it. In addition, the variable where we receive the parameter will have life during the execution of the function and will cease to exist when the function finishes its execution.

To call a function that has parameters, place the value of the parameter in parentheses. To call the example function you would have to write:

```
writeWelcome ("Alberto García")
```

When calling the function like this, the name parameter takes the value "Alberto García" and when writing the greeting on the screen it will write "Hello Alberto García" between <H1> tags.

The parameters can receive any type of data, numeric, textual, boolean or an object. We do not really specify the type of the parameter, so we must take special care when defining the actions we perform within the function and passing values to it, to ensure that everything is consistent with the types of data that we expect our variables or parameters to have.

Multiple parameters

A function can receive as many parameters as we want and to express it the names of the parameters are separated by commas, inside the parentheses. Let's quickly see the syntax for the function from before, but made so that it receives two parameters, the first the name to greet and the second the color of the text.

```
function writeWelcome (name, colorText) {
   document.write ("<FONT color = '" + colorText + "'>")
   document.write ("<H1> Hello" + name + "</H1>")
   document.write ("</FONT>")
}
```

We would call the function with this syntax. Between the brackets we will place the values of the parameters.

```
var myName = "Pepe"
var myColor = "red"
write Welcome (myName, myColor)
```

I have placed two variables in parentheses instead of two quoted texts. When we put variables between the parameters what we are actually passing to the function are the values that the variables contain and not the variables themselves.

Parameters are passed by value

In line with the use of parameters in our Javascript programs, we have to know that the parameters of the functions are passed by value. This means that we are passing values and not variables. In practice, even if we modify a parameter in a function, the original variable that we had passed will not change its value. It can be easily seen with an example.

```
function stepValue (myParameter) {
  myparameter = 32
  document.write ("I have changed the value to 32")
}
var myVariable = 5
stepValue (myVariable)
document.write ("the value of the variable is:" + myVariable)
```

In the example we have a function that receives a parameter and modifies the value of the parameter assigning it the value 32. We also have a variable, which we initialize to 5 and later we call the function passing this variable as a parameter. As within the function we modify the value of the parameter, it could happen that the original variable changes its value, but since the parameters do not modify the original value of the variables, it does not change its value.

In this way, once the function is executed, when printing the value of miVariable on screen, the number 5 will be printed, which is the original value of the variable, instead of 32, which was the value with which we had updated the parameter.

In Javascript you can only pass variables by value.

Return values in Javascript functions

We are learning about the use of functions in Javascript and at the moment we may have already realized the great importance they have in making programs more or less advanced. In this article of the Javascript Manual we will continue learning things about functions and specifically that with them you can also return values. In addition, we will see some interesting use case on the functions that can clarify a bit the scope of local and global variables.

Returning values in functions

Javascript functions can also return values. In fact, this is one of the most essential utilities of the functions, which we must know, not only in Javascript but in general in any programming language. So, by invoking a function, you can perform actions and offer a value as output.

For example, a function that calculates the square of a number will have that number as input and as the output it will have the value resulting from finding the square of that number. The data entry in the functions we saw earlier in the article on function parameters. Now we have to learn about the exit.

Let's look at an example of a function that calculates the mean of two numbers. The function will receive both numbers and return the value of the mean.

```
function media (value1, value2) {
  var result
  result = (value1 + value2) / 2
  return result
}
```

To specify the value that the function will return, the word return is used, followed by the value that you want to return. In this case, the content of the result variable is returned, which contains the calculated mean of the two numbers.

We may now ask ourselves how to receive a data returned by a function. Actually, in the source code of our programs we can invoke the functions in

the place that we want. When a function returns a value, the function call is simply replaced by that value it returns. So, to store a return value of a function, we have to assign the call to that function as content in a variable, and we would do that with the assignment operator =.

To illustrate this, you can see this example, which will call the mean () function and save the result of the mean in a variable and then print it on the page.

```
var my media
myAverage = mean (12.8)
document.write (myMedia)
```

Multiple return

Actually in Javascript functions can only return one value, so in principle we cannot make functions that return two different data.

Note: in practice nothing prevents us from having a function return more than one value, but since we can only return one thing, we would have to put all the values that we want to return in a data structure, such as an array. However, that would be a more or less advanced use that we are not going to see at the moment.

Now, although we can only return one data, in the same function we can place more than one return. As we say, we are only going to be able to return one thing, but depending on what has happened in the function it may be of one type or another, with some data or others.

In this function we can see an example of using multiple return. This is a function that returns 0 if the received parameter was even and the value of the parameter if it was odd.

```
function multipleReturn (number) {
  var remainder = number% 2
  if (remainder == 0)
    return 0
  else
    return number
}
```

To find out if a number is even we find the remainder of the division by dividing it by 2. If the remainder is zero it is that it was even and we return 0, otherwise -the number is odd- we return the received parameter.

Scope of variables in functions

Within the functions we can declare variables. On this matter we must know that all the variables declared in a function are local to that function, that is, they will only be valid during the execution of the function.

Note: Even if we think about it, we can realize that the parameters are like variables that are declared in the function's header and that are initialized when calling the function. The parameters are also local to the function and will be valid only when the function is running.

It could be the case that we can declare variables in functions that have the same name as a global variable to the page. So, inside the function, the variable that will be valid is the local variable and outside the function the global variable to the page will be valid.

On the other hand, if we do not declare the variables in the functions, it will be understood by javascript that we are referring to a global variable to the page, so if the variable is not created it creates it, but always global to the page instead of local to the function.

Let's look at the following code.

```
function variables_glogales_y_locales () {
  var local variable = 23
  globalGlobal = "qwerty"
}
```

In this case variableLocal is a variable that has been declared in the function, so it will be local to the function and will only be valid during its execution. On the other hand, variableGlobal has not been declared (because before

using it, the word var has not been used to declare it). In this case the variable variableGlobal is global to the whole page and will continue to exist even if the function ends its execution. Also, if the variable variableGlobal existed before calling the function, as a result of executing this function, a hypothetical value of that variable would be pounded and replaced by "qwerty".

Note: We can find more information about variable scope in a previous article.

Javascript function library

In all programming languages there are libraries of functions that serve to do different and very repetitive things when programming. Libraries in programming languages save you the trouble of writing common functions that programmers may usually need. A well-developed programming language will have a fair amount of them. Sometimes it is more difficult to know all the libraries well than to learn to program in the language.

Javascript contains a good number of functions in its libraries. As it is a language that works with objects, many of the libraries are implemented through objects. For example, math or string handling functions are implemented using Math and String objects. However, there are some functions that are not associated with any object and are the ones that we will see in this chapter, since we still do not know the objects and we will not need them to study them.

Built-in Javascript functions

These are the functions that Javascript makes available to programmers.

eval (string)

This function receives a character string and executes it as if it were a Javascript statement.

parseInt (string, base)

Receive a chain and a base. Returns a numeric value resulting from converting the string to a number in the indicated base.

parseFloat (string)

Converts the string to a number and returns it.

escape (character)

Returns a character that it receives as a parameter in an ISO Latin 1 encoding.

unescape (character)

It does the exact opposite of the escape function.

isNaN (number)

Returns a boolean depending on what it receives per parameter. If it is not a number it returns true, if it is a number it returns false.

The libraries that are implemented through objects and those of the browser handling, which are also managed with objects, we will see later.

Note: We don't want to mislead people with this short list of native Javascript functions. There really are many other functions that we will see throughout this manual, what happens is that they are associated with objects. For example, as we have pointed out, there are character string functions, which are associated with string objects, functions for working with advanced mathematical calculations, which are associated with the Math class, functions for working with the object of the browser window, with the document, etc.

Examples of using the functions incorporated in Javascript

So far we have simply known a list of the native functions of the Javascript language. Now we can see several examples of the use of native Javascript functions, which we have available in any browser and in any version of Javascript.

We will see three functions from different fields that are quite fundamental in the usual work with this language, explained through examples.

Eval function

This function is very important, so much so that there are some Javascript applications that could not be performed if we do not use it. Its use is very simple, but it may be a little more complex to understand in which cases to use it because sometimes its application is a little subtle.

With current knowledge we cannot make a very complicated example, but at least we can see the function working. We are going to use it in a slightly strange and quite useless statement, but if we can understand it, we will also understand the function eval.

```
var myText = "3 + 5"
eval ("document.write (" + myText + ")")
```

First we create a variable with a text, in the next line we use the function eval and as a parameter we pass a javascript instruction to write to the screen. If we concatenate the strings that are inside the parentheses of the function eval, we are left with this.

```
document.write (3 + 5)
```

The eval function executes the instruction passed to it by parameter, so it will execute this statement, which will result in an 8 being written to the web page. First the sum in parentheses is solved, so we get the 8 and then the write on screen instruction is executed.

ParseInt function

This function receives a number, written as a string, and a number that indicates a base. You can actually receive other types of variables, since variables have no type in Javascript, but it is usually used by passing a string to convert the text variable to a number.

The different bases that the function can receive are 2, 8, 10 and 16. If we do not pass any value as a base, the function interprets the base as decimal. The value returned by the function always has base 10, so if the base is not 10 it converts the number to that base before returning it.

Let's look at a series of calls to the parseInt function to see what it returns and to understand the function a little more.

document.write (parseInt ("34"))

Returns the number 34

document.write (parseInt ("101011", 2))

Returns the number 43

document.write (parseInt ("34", 8))

Returns the number 28

document.write (parseInt ("3F", 16))

Returns the number 63

This function is used in practice for a lot of different things in number handling, for example getting the integer part of a decimal.

document.write (parseInt ("3.38"))

Returns the number 3

It is also very common to use it to know if a variable is numeric, because if we pass a text to the function that is not numeric it will return NaN (Not a Number) which means that it is not a Number.

document.write (parseInt ("develoloweb.com"))

Returns the NaN number

This same example is interesting with a modification, because if we pass a combination of letters and numbers it will give us the following.

document.write (parseInt ("16XX3U"))

Returns the number 16

document.write (parseInt ("TG45"))

Returns the NaN number

As you can see, the function tries to convert the string to number and if it can't it returns NaN.

All of these somewhat unrelated examples of how parseInt works will be reviewed later in more practical examples when dealing with working with forms.

IsNaN function

This function returns a boolean depending on whether it receives a number or not. The only thing you can receive is a number or the expression NaN. If it

receives a NaN it returns true and if it receives a number it returns false. It is a very simple function to understand and use.

The function usually works in combination with the parseInt or parseFloat function, to know if what these two functions return is a number or not.

miInteger = parseInt ("A3.6")

isNaN (myInteger)

In the first line we assign the variable miInteger the result of trying to convert the text A3.6 to an integer. Since this text cannot be converted to a number, the parseInt function returns NaN. The second line checks if the previous variable is NaN and as if it is, it returns true.

miFloat = parseFloat ("4.7")

isNaN (miFloat)

In this example we convert a text to a number with decimals. The text converts perfectly because it corresponds to a number. When receiving a number the function isNaN returns a false.

We have a very interesting Javascript Workshop that has been carried out to strengthen the knowledge of these chapters. It is a script to validate a form field so that we know for sure that within the field there is always an integer. It can be very interesting to read it now, since we use the isNaN () and parseInt () functions. See the workshop

We hope that the examples seen in this article have been interesting. However, as we noted earlier, there are quite a few other native Javascript functions that we should be aware of, but which are associated with native Javascript classes and objects. But before moving on to that point we want to offer a small basic guide for working with object-oriented programming in Javascript.

Arrays in Javascript

In programming languages there are special data structures that help us store more complex information than simple variables. A typical structure in all languages is the Array, which is like a variable where we can enter several values, instead of just one as it happens with normal variables.

Arrays allow us to save several variables and access them independently, it is like having a variable with different compartments where we can enter different data. For this we use an index that allows us to specify the compartment or position to which we are referring.

Note: The arrays were introduced in Javascript versions 1.1 or higher, that is, we can only use them from browsers 3.0. For older browsers you can simulate the array using object-oriented programming syntax, but the truth is that currently this limitation should not concern us. Furthermore, given the complexity of the task of simulating an array by means of objects, at least at the time we meet and the few occasions when we will need it, we think that it is better to forget about that matter and simply work with arrays usually. So in this article and the following we are going to see how to use the authentic Javascript array.

Creation of javascript arrays

The first step in using an array is to create it. For this we use a Javascript object already implemented in the browser. We will now see a topic to explain what object orientation is, although it will not be necessary to understand the use of arrays. This is the statement to create an array object:

var myArray = new Array ()

This creates an array on the page that is running. The array is created without any content, that is, it will not have any boxes or compartments created. We can also create the Javascript array specifying the number of compartments it will have.

var myArray = new Array (10)

In this case, we indicate that the array will have 10 positions, that is, 10 boxes where to save data.

It is important to note that the word Array in Javascript code is written with the first letter capitalized. As in uppercase and lowercase javascript if they matter, if we write it in lowercase it will not work.

Whether or not the number of boxes in the javascript array is indicated, we can enter any data in the array. If the box is created it is simply entered and if the box was not created it is created and then the data is entered, so the final result is the same. This checkbox creation is dynamic and occurs at the same time that the scripts are executed. Let's see below how to enter values in our arrays.

myArray [0] = 290

myArray [1] = 97

myArray [2] = 127

They are introduced by indicating in square brackets the index of the position where we wanted to save the data. In this case we enter 290 in position 0, 97 in position 1 and 127 in position 2.

Arrays in Javascript always start at position 0, so an array that has 10 positions, for example, will have boxes from 0 to 9. To collect data from an array we do the same: putting the index of position a in square brackets which we want to access. Let's see how the content of an array would be printed on the screen.

```
var myArray = new Array (3)

myArray [0] = 155
myArray [1] = 4
myArray [2] = 499

for (i = 0; i <3; i ++) {
  document.write ("Position" + i + "of the array:" + myArray [i])
  document.write ("<br>")
}
```

We have created an array with three positions, then we have entered a value in each of the positions of the array and finally we have printed them. In general, traversing arrays to print their positions, or anything else, is done using loops. In this case we use a FOR loop that goes from 0 to 2.

We can see the example running on another page.

Data types in arrays

In the boxes of the arrays we can save data of any type. We can see an array where we enter character data.

miArray [0] = "Hello"

myArray [1] = "a"

miArray [2] = "all"

Even in Javascript we can save different types of data in the boxes of the same array. That is, we can enter numbers in some boxes, texts in others, Booleans or anything else we want.

miArray [0] = "develoloweb.com"

myArray [1] = 1275

miArray [1] = 0.78

myArray [2] = true

Array declaration and summary initialization

In Javascript we have at our disposal a summarized way to declare an array and load values in the same step. Let's look at the following code:

var arrayRapido = [12,45, "array initialized in its declaration"]

As you can see, we are defining a variable called fastArray and we are indicating in the square brackets several values separated by commas. This is the same as having declared the array with the Array () function and then loading the values one by one.

Array length

In the previous article in the Javascript Manual we started to explain the concept of array and its use in Javascript. In this article, we will continue with the topic, showing the use of its length property.

All arrays in javascript, apart from storing the value of each of their boxes, also store the number of positions they have. To do this they use a property of the array object, the length property. We will see in objects what a property is, but in our case we can imagine that it is like a variable, in addition to the positions, that stores a number equal to the number of boxes that the array has.

To access a property of an object, the dot operator must be used. The name of the array that we want to access is written to the number of positions it has, without brackets or parentheses, followed by a period and the word length.

var myArray = new Array ()

myArray [0] = 155

myArray [1] = 499

myArray [2] = 65

document.write ("Array length:" + myArray.length)

This code would print on the screen the number of positions of the array, which in this case is 3. We remember that an array with 3 positions ranges from position 0 to 2.

It is very common for the length property to be used to traverse an array through all its positions. To illustrate it, we are going to see an example of a

walk through this array to show its values.

```
for (i = 0; i <myArray.length; i ++) {
  document.write (myArray [i])
}
```

Note that the for loop is executed whenever i is less than the length of the array, taken from its length property.

The following example will help us to better understand the routes through the arrays, the operation of the length property and the dynamic creation of new positions. We are going to create an array with 2 positions and fill its value. Later we will introduce a value in position 5 of the array. Finally we will print all the positions of the array to see what happens.

```
var myArray = new Array (2)

miArray [0] = "Colombia"
miArray [1] = "United States"

miArray [5] = "Brazil"

for (i = 0; i <myArray.length; i ++) {
  document.write ("Position" + i + "of the array:" + myArray [i])
  document.write ("<br>")
}
```

The example is simple. It can be seen that we walk through the array from 0 to the number of positions in the array (indicated by the length property). Along the way we are printing the number of the position followed by the content of the array in that position. But we can have a doubt when we ask

ourselves what will be the number of elements in this array, since we had declared it with 2 and then we have introduced a third in position 5. When we see the exit of the program we will be able to answer our questions. It will look something like this:

Array position 0: Colombia

Array position 1: United States

Array position 2: null

Array position 3: null

Array position 4: null

Array position 5: Brazil

It can be clearly seen that the number of positions is 6, from 0 to 5. What has happened is that when entering a data in position 5, all the boxes that were not created until the fifth are also created.

Positions 2 through 4 are uninitialized. In this case our browser has written the word null to express this, but other browsers may use the word undefined. We will see later what this null is and where we can use it, the important thing now is that you understand how arrays work and use them correctly.

Multidimensional arrays in Javascript

As we are seeing, arrays are quite important in Javascript and also in most programming languages. Specifically, we have already learned to create arrays and use them in previous articles in the Javascript Manual. But we still have some important things to explain, such as arrays of various dimensions.

Multidimensional arrays are a data structure that stores values in more than one dimension. The arrays that we have seen so far store values in one dimension, so we only use an index to access positions. 2-dimensional arrays store their values, so to speak, in rows and columns and therefore we will need two indexes to access each of their positions.

In other words, a multidimensional array is like a container that stores more values for each position, that is, as if the elements of the array were in turn other arrays.

In Javascript there is no real multidimensional array-object. To use these structures we can define arrays that where in each of their positions there will be another array. In our programs we can use arrays of any dimension, we will see below how to work with two-dimensional arrays, which will be the most common.

In this example we are going to create a two-dimensional array where we will have cities on the one hand and the average temperature in each one during the winter months on the other.

var city_media_temperatures0 = new Array (3)

city_media_temperatures0 [0] = 12

city_media_temperatures0 [1] = 10

city_media_temperatures0 [2] = 11

var city_media_temperatures1 = new Array (3)

city_media_temperatures1 [0] = 5

city_media_temperatures1 [1] = 0

city_media_temperatures1 [2] = 2

var city_media_temperatures2 = new Array (3)

city_media_temperatures2 [0] = 10

city_media_temperatures2 [1] = 8

city_media_temperatures2 [2] = 10

With the previous lines we have created three 1-dimensional arrays and three elements, like the ones we already knew. Now we will create a new array of three elements and we will introduce into each of its boxes the arrays created previously, with which we will have an array of arrays, that is, a 2-dimensional array.

var cities_temperatures = new Array (3)

temperatures_city [0] = average_city_city0

city_temperatures [1] = city_average_ temperatures1

city_temperatures [2] = city_average_ temperatures2

We see that to introduce the entire array we refer to it without parentheses or brackets, but only with its name. The array temperatures_cities is our two-dimensional array.

It's also interesting to see how a tour of a two-dimensional array is performed. For this we have to make a loop that goes through each of the squares of the two-dimensional array and within these make a new path for

each of its internal squares. That is, a tour of one array within another.

The method of looping through another is to put one loop inside another, which is called a nested loop. In this example we are going to put one FOR loop inside another. In addition, we are going to write the results in a table, which will complicate the script a bit, but this way we will be able to see how to build a table from Javascript as we make the nested tour of the loop.

```
document.write ("<table width = 200 border = 1 cellpadding = 1 cellspacing = 1>");
for (i = 0; i <cities_temperatures.length; i ++) {
  document.write ("<tr>")
  document.write ("<td> <b> City" + i + "</b> </td>")
  for (j = 0; j <temperatures_cities [i] .length; j ++) {
    document.write ("<td>" + cities_temperatures [i] [j] + "</td>")
  }
  document.write ("</tr>")
}
document.write ("</table>")
```

This script is a little more complex than those seen previously. The first action is to write the table header, that is, the <TABLE> tag along with its attributes. With the first loop we make a journey to the first dimension of the array and use the variable i to keep track of the current position. For each iteration of this loop we write a row and to start the row we open the <TR> tag. In addition, we write in a box the number of the city that we are visiting at that time. Later we put another loop that goes through each of the cells of the array in its second dimension and write the temperature of the current city in each of the months, inside its <TD> tag. Once the second loop ends, all three temperatures have been printed and therefore the row is finished. The first loop continues repeating until all cities are printed and once finished we close the table.

Note: You may have noticed that sometimes generating HTML code from Javascript becomes complex. But the problem is not only that the code is

difficult to produce, but the worst thing is that you create a code that is difficult to maintain, in which both the part of the programming in Javascript is mixed with the part of the presentation in HTML. What you have also seen is just a very simple code, with a really elementary table, imagine what would happen when the table or the data were more complex. Fortunately, there are better ways to generate HTML output than we have seen now, although it is a bit advanced at the moment. Anyway, we leave you a link to the manual of the Javascript Handlebars templates system, which is a simple library alternative to generate HTML output from Javascript.

We can see the running example and examine the entire script code.

Array initialization

To finish with the topic of arrays, we are going to see a way to initialize its values at the same time that we declare it, so we can carry out the process of entering values in each of the positions of the array in a faster way.

The normal method of creating an array we saw was through the Array object, putting the number of boxes in the array in parentheses or not putting anything, so that the array is created without any position. To introduce values to an array it is done the same, but putting the values with which we want to fill the cells separated by commas between the parentheses. Let's see it with an example that creates an array with the names of the days of the week.

var diasWeek = new Array ("Monday", "Tuesday", "Wednesday,", "Thursday", "Friday", "Saturday", "Sunday")

The array is created with 7 boxes, from 0 to 6 and in each box the corresponding day of the week is written (In quotes because it is a text).

Now we are going to see something more complicated, it is about declaring the two-dimensional array that we used before for the temperatures of the cities in the months in a single line, entering the values at the same time.

var cities_temperatures = new Array (new Array (12,10,11), new Array (5,0,2), new Array (10,8,10))

In the example we introduce in each box of the array another array that has as values the temperatures of a city in each month.

Javascript still has a more summarized way than the one we just saw, which we explained in the first article where we covered arrays. To do this we simply put the data of the array we are creating in square brackets. To finish we will show an example on how to use this syntax to declare arrays of more

than one dimension.

var arrayMuchasDimensiones = [1, ["hello", "que", "tal", ["these", "we are", "I am"], ["good", "bad"], "I just"], 2, 5];

In this example we have created a very uneven array, because it has boxes with content of simple integers and others with content of string and others that are other arrays. We could access some of their boxes and display their values like this:

alert (arrayMuchasDimensiones [0])

alert (arrayMuchasDimensiones [1] [2])

alert (arrayMuchasDimensiones [1] [3] [1])

General introduction to objects in Javascript

We are going to introduce ourselves to a very important Javascript topic such as objects. It is a topic that we have not yet seen and on which we are going to constantly deal with since most of the things in Javascript, even the simplest, we are going to do through the handling of objects. In fact, in the examples made so far we have made great efforts not to use objects and even so we have used them on occasion, since it is very difficult to find examples in Javascript that, although simple, do not make use of them.

Object Oriented Programming (OOP) represents a new way of thinking when making a program. Javascript is not a pure object-oriented programming language because, although it uses objects many times, we don't need to program all of our programs based on them. In fact, what we are generally going to do with Javascript is use objects and not so much object oriented programming. Therefore, the way of programming is not going to change much compared to what we have seen so far in the Javascript Manual. In summary, what we have seen so far regarding syntax, functions, etc. it is still perfectly valid and can be used just as indicated. We are only going to learn a kind of new structure such as objects.

Note: To start getting a bit soaked about objects we have a small article published in DesarrolloWeb about object-oriented programming. It would be highly recommended that you read it, because several concepts are explained in which we will not go into so much detail. If you already know the POO, continue reading without pause, but if you want to go deeper, remember that we also have a complete Object Orientation Manual. If you like to watch videos we also recommend the What are the objects class taught in the Video Programming Course.

What is an object

Although we are not going to go into detail with the concetos, as they are very well explained in references that we have already indicated, objects are a programming language tool in which two fundamental things come together: data and functionality. Every computer program basically deals with those two things in some way. With what we have seen so far we had the data in variables and the functionality in functions, right? because in the world of objects, both data and functionality are in the same structure, the object.

The thing is that now you need to learn new names with which to refer to the data and functionality grouped into an object:

Properties: In objects the properties refer to the data

Methods: In objects, methods refer to functionality

Imagine you have a button object (a browser button, something you can press to perform an action). The button has a written text, because that text would be a data and therefore we would call it property. Another property of a button would be whether or not it is activated. On the other hand, a button could have associated functionality, which would be in a method, such as processing the action of a click. Imagine something more generic like a phone. The phone may have properties such as make, model, operating system, and methods such as turning on, off, calling a number, etc.

In pure object-oriented programming languages, such as Java, you always have to program based on objects. To program you would have to create "classes", which are a kind of "molds" from which objects are created. The program would solve any need by creating objects based on those molds (classes), there being several (tens, hundreds or thousands) of objects of different classes. The objects would have to collaborate with each other to solve any type of action, just as in systems like an airplane there are various objects (the engine, propellers, controls ...) that collaborate with each other to solve the need to carry passengers or merchandise on trips aerial.

However, as we have been saying, in Javascript it is not so much object-oriented programming, but using objects. Many times they will be objects already created by the browser itself (the browser window, an HTML document being displayed, an image or a form within that HTML document, etc.), and other times they will be objects created by yourself or by others. developers that help you do specific things. Therefore, what we want to know to get started is the syntax you need to use objects, basically access their properties and execute their methods.

Note: To know what the browser objects are, which we have available in Javascript to solve the needs of web pages, you have to read the manual on working with Javascript to use and manipulate browser resources.

How to access object properties and methods

In Javascript we can access the properties and methods of objects in a similar way to what we do in other programming languages, with the operator dot (".").

The properties are accessed by placing the name of the object followed by a period and the name of the property to be accessed. In this way:

```
myObject.myProperty
```

To call the methods we use a similar syntax, but putting the parameters that we pass to the methods in parentheses. As follows:

```
myObject.myMethod (parameter1, parameter2)
```

If the method does not receive parameters we put the parentheses too, but with nothing inside.

```
myObject.myMethod ()
```

How to create objects

As we have said, most of the objects with which you are going to work in Javascript in order to create interaction, effects and diverse behaviors on web pages, they give you ready-made. The browser itself offers them to you so you simply have to use them. That's study material from the Javascript Manual and browser objects. Having clarified that point, it should be noted that Javascript is somewhat particular when creating objects, basically because traditionally there is no "class" concept.

To be more exact, in Javascript, ES5, classes are created by means of functions and with the new operator you create objects from those functions, but there are no classes like the ones we know in other more traditional languages.

Note: Classes already exist in ES6 and Javascript is capable of generating classes and from them producing objects, like other languages. You get more information in the ES6 Manual.

The other alternative to create objects in Javascript is by means of object literals, which are nothing more than the definition of the object by means of code enclosed in curly braces, indicating their properties or methods as is.

```
{
  name: 'Miguel Angel Alvarez',
  SitioWeb: 'DesarrolloWeb.com'
}
```

In the previous code we have only defined properties, but we have other articles where you can see how to define methods as well. To know more about object literals we recommend reading the article on Javascript object literals, where we will explain more carefully this usual syntax for creating objects in this language.

In traditional Javascript we have said that classes do not exist, but we can create instances of objects from functions, as we will see in the next point.

Create and instantiate objects from functions

For those who do not know, instantiating an object is the action of creating an instance of a class, so that you can work with it later. Class is the definition of the characteristics and functionalities of an object. Classes do not work directly, these are only definitions. To work with a class we must have an instantiated object of that class. We remember that in Javascript there are no classes, but we can use functions.

This simple function could be used as a template to build objects of the Person class:

```
function Person (name) {
  this.name = name;
}
```

You will notice that we are using the word "this" inside. That word is a reference to the object to be created with this function. In javascript to create an object from a function the new statement is used, in this way.

```
var miguel = new Persona ('Miguel Angel Alvarez');
```

In a variable we call "miguel" I assign a new (new) instance of the Person class. The parentheses are filled with the data that the class needs to initialize the object, if there are no parameters to enter, the parentheses are placed empty. Actually what you do when you create an object is to call the function that builds it and the function itself will take care of initializing the properties of the object (which is what the "this" reference uses).

Modify property

When the property is already present, Object.defineProperty () will try to

modify it according to the values in the property descriptor and adjust the current object. If the property descriptor in the object has set the configurable value to false, the property is not set and can not be set Modify them (only modify the value of the writable key to false [and vice versa]); it is not possible to switch between properties of properties (access properties and data properties) when the property is not adjustable.

The TypeError error will be called when you try to change non-adjustable properties (except for a writable value as shown above).

The key is writable

When you set the value of the writable key to false, the property is not writable, that is, the values can not be re-assigned:

```
var o = {};
Object.defineProperty (o, 'a', {
  value: 1,
  enumerable: true
});
Object.defineProperty (o, 'b', {
 value: 2,
 enumerable: false
});
Object.defineProperty (o, 'c', {
  value: 3
}); // property not enumerated
o.d = 4; // statistic property because we created it via
        // Assign value to a property directly
```

```
for (var i in o) {
  console.log (i);
}
// 'a' and 'd' (unordered)

Object.keys (o); // ['a', 'd']

o.propertyIsEnumerable ('a'); // true
o.propertyIsEnumerable ('b'); // false
o.propertyIsEnumerable ('c'); // false
```

The key is configurable

The configurable key controls whether the property can be deleted from the object (via the delete parameter), and whether the rest of the keys can be changed (except that the value of the key is writable to false):

```
var o = {};
Object.defineProperty (o, 'a', {
  get: function () {return 1; },
  configurable: false
});

Object.defineProperty (o, 'a', {
  configurable: true
}); // throws a TypeError
```

```
Object.defineProperty (o, 'a', {
  enumerable: true
}); // throws a TypeError
Object.defineProperty (o, 'a', {
  set: function () {}
}); // throws a TypeError (set was undefined previously)
Object.defineProperty (o, 'a', {
  get: function () {return 1; }
}); // throws a TypeError
Object.defineProperty (o, 'a', {
  value: 12
}); // throws a TypeError

console.log (o.a); // 1
delete o.a; // Nothing will happen
console.log (o.a); // 1
```

If the configurable key value of the o.a property is true, none of the previous errors will be thrown, and the property will eventually be deleted.

Add the default attributes and values of the keys

It is important to take into account the default values of the keys. There is a big difference between assigning a value to a property of the object directly and using the Object.defineProperty () function as shown in the following example:

```
var o = {};

oa = 1;
// Rewards expression
Object.defineProperty (o, 'a', {
  value: 1,
  writable: true,
  configurable: true,
  enumerable: true
});
"But in return
Object.defineProperty (o, 'a', {value: 1});
// Rewards
Object.defineProperty (o, 'a', {
  value: 1,
  writable: false,
  configurable: false,
  enumerable: false
});
```

Custom getter and setter functions

The following example demonstrates how to create an object that automatically values its assigned values. When you set a value for the property temperature, it is added to the archive:

```
function Archiver () {
  var temperature = null;
  var archive = [];

  Object.defineProperty (this, 'temperature', {
    get: function () {
      console.log ('get!');
      return temperature;
    },
    set: function (value) {
      temperature = value;
      archive.push ({val: temperature});
    }
  });

  this.getArchive = function () {return archive; };
}

var arc = new Archiver ();
arc.temperature; // 'get!'
arc.temperature = 11;
arc.temperature = 13;
arc.getArchive (); // [[val: 11}, {val: 13}]
```

Another example of the getter and setter functions:

```
var pattern = {
    get: function () {
        return 'I always return this string,' +
               'whatever you have assigned';
    },
    set: function () {
        this.myname = 'this is my name string';
    }
};


function TestDefineSetAndGet () {
    Object.defineProperty (this, 'myproperty', pattern);
}


var instance = new TestDefineSetAndGet ();
instance.myproperty = 'test';
console.log (instance.myproperty);
// I always return this string, whatever you have assigned

console.log (instance.myname); // this is my name string
```

Supports all browsers

- **Object.defineProperties ()**

The Object.defineProperties () function defines new properties on an object directly, modifies properties that already exist in an object, and then reopens that object.

General structure

Object.defineProperties (obj, props)

obj

The object in which you want to define properties or modify them.

props

An object with its own enumerated properties specifying the property descriptors that will be added to or modified by the object and which will be associated with attribute names. There are two types of descriptors: data descriptors and accessor descriptors, see the Object.defineProperty () function page for more details.

Data descriptors and access descriptors are objects, sharing the following required keys:

* configurable: If the value of this key is true, you can modify the property descriptor (not the value of the property) and the property can be deleted

from the object. The default value is false.

enumerable: If the value of this key is true, this property will appear when enumerating the object's enumeration properties. The default value is false.

The following optional keys can exist in metadata:

* value: Specifies the value associated with the property, and any value can be allowed in JavaScript (such as numbers, functions, objects, etc.). The default value is undefined.

writable: If the value of this key is true, the value associated with this property can be modified by using the assignment coefficients. The default value is false.

The following optional keys can be found in access descriptors:

* get: The function that will be set as a getter function for this property, or the undefined value if there is no getter function associated with this property; the value returned by this function will be used as a value for the property. The default value is undefined.

set: The function that will be set as a setter function for this property, or the undefined value if there is no setter function associated with this property; this function accepts one argument that is the new value that the user attempted to attribute to this property. The default value is undefined.

Returned value

The object that passed to the function.

the description

The Object.defineProperties () function defines all properties on the obj object, which are dependent on the object (and must also be enumerated).

Examples

An example of defining properties property1 and property2 on object obj, one of these properties is writeable (by assigning the true value of the writable key) and another non-writable (by assigning the false value to the writable key):

```
var obj = {};
Object.defineProperties (obj, {
  'property1': {
    value: true,
    writable: true
  },
  'property2': {
    value: 'Hello',
    writable: false
  }
  // ...
});
```

Reduced support for browsers

Assuming that the operating environment is not modified, as all the names and properties indicate their initial values, the following function is equivalent to the object.defineProperties () function:

```
function defineProperties (obj, properties) {
  function convertToDescriptor (desc) {
    function hasProperty (obj, prop) {
      return Object.prototype.hasOwnProperty.call (obj, prop);
    }

    function isCallable (v) {
      // NB: modify as necessary if other values than functions are callable.
      return typeof v === 'function';
    }

    if (typeof desc! == 'object' || desc === null)
      throw new TypeError ('bad desc');

    var d = {};

    if (hasProperty (desc, 'enumerable'))
      d.enumerable = !! desc.enumerable;
    if (hasProperty (desc, 'configurable'))
```

```
    d.configurable = !! desc.configurable;
  if (hasProperty (desc, 'value'))
    d.value = desc.value;
  if (hasProperty (desc, 'writable'))
    d.writable = !! desc.writable;
  if (hasProperty (desc, 'get')) {
    var g = desc.get;

    if (! isCallable (g) && typeof g! == 'undefined')
      throw new TypeError ('bad get');
    d.get = g;
  }
  if (hasProperty (desc, 'set')) {
    var s = desc.set;
    if (! isCallable (s) && typeof s! == 'undefined')
      throw new TypeError ('bad set');
    d.set = s;
  }

  if (('get' in d || 'set' in d) && ('value' in d || 'writable' in d))
    throw new TypeError ('identity-confused descriptor');

  return d;
}

if (typeof obj! == 'object' || obj === null)
  throw new TypeError ('bad obj');
```

```
  properties = Object (properties);

  var keys = Object.keys (properties);
  var descs = [];

  for (var i = 0; i <keys.length; i ++)
    descs.push ([keys [i], convertToDescriptor (properties [keys [i]])]);

  for (var i = 0; i <descs.length; i ++)
    Object.defineProperty (obj, descs [i] [0], descs [i] [1]);

  return obj;
}
```

Supports all browsers

- **Object.entries ()**

The Object.entries () function returns a matrix that contains the statistic properties of an object in the form of [key, value] pairs in the same order that the loop for ... in.

The difference between this function and the loop for ... is that the for ... in

loop will override the statistically significant properties in the prototype string as well.

General structure

Object.entries (obj)

obj

The object whose dependent properties will be returned as pairs [key, value].

Returned value

A matrix containing the values of the dependent properties of the obj object in the form of pairs [key, value].

Examples

Example of a simple object:

```
const obj = {foo: 'bar', baz: 42};
console.log (Object.entries (obj)); // ['foo', 'bar'], ['baz', 42]]
```

Example of an arithmetic-like object:

```
const obj = {0: 'a', 1: 'b', 2: 'c'};
console.log (Object.entries (obj)); (I), [b], [ii], [c]]]
```

An example of an object similar to matrices but the order of keys is random:

```
const anObj = {100: 'a', 2: 'b', 7: 'c'};
console.log (Object.entries (anObj)); ['(Ii),' b '], [7', 'c'], [100 ',' a ']]
```

The getFoo property is a non-statistic property in the myObj object:

```
const myObj = Object.create ({}, {getFoo: {value () {return this.foo;}}});
myObj.foo = 'bar';
console.log (Object.entries (myObj)); // [['foo', 'bar']]
```

Note that if you pass a broker that does not represent an object, it will be converted to an object, as in the following text string:

```
console.log (Object.entries ('foo')); (I), [o]], [ii], 'o']]
```

But an empty matrix will be returned for any initial data type, because the initial values do not have any attributes associated with them:

```
console.log (Object.entries (100)); // []
console.log (Object.entries (true)); // []
```

Passing object properties and values through the loop for ... of:

```
const obj = {a: 5, b: 7, c: 9};
for (const [key, value] of Object.entries (obj)) {
  console.log (`$ {key} $ {value}`); // "a 5", "b 7", "c 9"
}
```

Or by using matrix functions:

```
Object.entries (obj) .forEach (([key, value]) => {
console.log (`$ {key} $ {value}`); // "a 5", "b 7", "c 9"
});
```

The Object.entries () function can be used to convert an object of type Object to Map:

```
const obj = {foo: 'bar', baz: 42};
const map = new Map (Object.entries (obj));
console.log (map); // Map {foo: "bar", baz: 42}
```

Reduced support for browsers

To add support for the Object.entries () function in environments that you do not support, you can use the following code:

```
if (! Object.entries)
  Object.entries = function (obj) {
    var ownProps = Object.keys (obj),
        i = ownProps.length,
        resArray = new Array (i); // preallocate the Array
    while (i--)
      resArray [i] = [ownProps [i], obj [ownProps [i]]];

    return resArray;
  };
```

If you want to support IE browsers that are released less than 9, you must provide an alternative function for the Object.keys function (which you will find on its page).

- Object.freeze ()

The Object.freeze () function blocks an object, prevents the addition of new properties to it, prevents the deletion of properties in it, and prevents the modification of the value, scalability, scalability, or writeability of its properties; «Frozen».

General structure

Object.freeze (obj)

obj

The object that will freeze.

Returned value

Frozen object.

the description

Any properties of frozen objects can not be added or deleted; any attempt to do so will either fail silently or cause the TypeError exception (this error will usually be in the strict style).

You can not change the values of attributes that have data, getter and setter functions, note that the values of objects assigned to attributes can still be modified unless they are also frozen.

Arrays can be frozen like objects, as we can not modify the values of their elements, or add or delete elements from them.

Note that in ECMAScript 5 the use of this function on a non-object medium would result in a TypeError throw, but starting with ECMAScript 2015 (ie

ES6), media that do not represent objects will be treated as frozen objects, ie their value will be returned as is :

```
> Object.freeze (1)
TypeError: 1 is not an object // ES5


> Object.freeze (1)
1 // ES2015
```

Examples

Freeze objects

In the following example, we will create an object named obj, add new properties, modify properties in it, and delete the other:

```
var obj = {
  prop: function () {},
  foo: 'bar'
};

obj.foo = 'baz';
obj.lumpy = 'woof';
delete obj.prop;
```

Note that the Object.freeze () function will return the frozen object, but it will also freeze the original object, so it is not necessary to save the returned object from the function to freeze the original object:

```
var o = Object.freeze (obj);

o === obj; // true
Object.isFrozen (obj); // === true
```

Note that the process of modifying attribute values or adding new properties will fail silently, but if the strict style is triggered, the TypeError error will be called:

```
obj.foo = 'quux'; // Nothing will happen
// The new property will not be added
obj.quaxxor = 'the friendly duck';

function fail () {
  'use strict';
  obj.foo = 'sparky'; // TypeError
  delete obj.quaxxor; // TypeError
  obj.sparky = 'arf'; // TypeError
}

fail ();
```

Attempting to make modifications using the Object.defineProperty () function will cause the TypeError error to be thrown:

```
Object.defineProperty (obj, 'ohai', {value: 17}); // TypeError
Object.defineProperty (obj, 'foo', {value: 'eit'}); // TypeError
```

It is impossible to modify the value of the prototype property, and both of the following expressions will throw the TypeError error:

Object.setPrototypeOf (obj, {x: 20})

obj .__ proto__ = {x: 20}

Freeze matrices

We will create an array named a, then freeze it and try to modify its values, and those attempts will fail silently; if we are in the strict pattern, the TypeError error will be called:

```
let a = [0];
Object.freeze (a); // The matrix can no longer be modified

a [0] = 1; // Will fail silently
a.push (2); // Will fail silently

// The TypeError error will be called
function fail () {
  "use strict"
  a [0] = 1;
  a.push (2);
}

fail ();
```

Deep freezing

It is true that the frozen object will not be adjustable, but it is not necessarily "fixed"; in the following example, the freeze will be superficial:

```
obj1 = {
  internal: {}
};

Object.freeze (obj1);
obj1.internal.a = 'aValue';

obj1.internal.a // 'aValue'
```

In order to make the object dormant, each of the properties of that object must be frozen. This is called "deep freeze":

```
function deepFreeze (obj) {

  // Get the names of properties in the object
  var propNames = Object.getOwnPropertyNames (obj);

  // Freeze those properties before freezing the object
  propNames.forEach (function (name) {
```

```
    var prop = obj [name];

    // Freeze the property if it is an object
    if (typeof prop == 'object' && prop! == null)
      deepFreeze (prop);
  });

  // Freeze the object itself
  return Object.freeze (obj);
}

obj2 = {
  internal: {}
};

deepFreeze (obj2);
obj2.internal.a = 'anotherValue';
obj2.internal.a; // undefined
```

These general attributes return a simple value, they do not represent the functions or attributes of an object.

- Infinity
- NaN
- Undefined
- Null

1.1 Infinity

The Infinity general property is a numerical value representing infinity.

Infinity property attributes

- Writeable - no
- Statistically - no
- Adjustable - No

General structure

```
Infinity
```

the description

The Infinity property is a property of the global object, that is, it is a variable in the public domain.

The value of the Infinity property is Number.POSITIVE_INFINITY, and the value of the Infinity property is greater than any other number. This value follows the behavior of the infinity value in mathematics. For example, the result of multiplying any number in Infinity is Infinity, and the output of any number on Infinity is 0.

Examples

The following examples show the result of mathematical operations on the value of Infinity:

```
console.log (Infinity); // Infinity
```

```
console.log (Infinity + 1); // Infinity
console.log (Math.pow (10, 1000)); // Infinity
console.log (Math.log (0)); // -Infinity
console.log (1 / Infinity); // 0
```

1.2 NaN

The public property NaN is a value that does not represent a number (an abbreviation for the Not-A-Number).

The properties of the NaN property

- Writeable – no
- Statistically – no
- Adjustable - No

General structure

```
NaN
```

the description

The NaN property is a property of the global object, that is, it is a variable in

the public domain.

The initial value of the NaN property is Number.NaN. The NaN property is not configurable and is not writable in modern browsers, but try to avoid writing it.

The value of NaN is rarely used in programs, but it is the value that will be returned when the functions of the Math object (such as Math.sqrt (-1)) fail or when a numeric value (such as parseInt ("blabla") fails).

Test if the value is NaN

The value of NaN (via the ==,! =, === and != ==) parameters is not equal to any other value, including the other NaN values; so use the Number.isNaN () or isNaN) To see if the value involved is NaN, or try to compare the value with itself, the NaN value is the only value that does not equal itself.

```
NaN === NaN; // false
Number.NaN === NaN; // false
isNaN (NaN); // true
isNaN (Number.NaN); // true

function valueIsNaN (v) {return v! == v; }
valueIsNaN (1); // false
valueIsNaN (NaN); // true
valueIsNaN (Number.NaN); // true
```

undefined in JavaScript

The undefined public property represents the undefined initial value in JavaScript.

The property attributes are undefined

- **Writeable – no**
- **Statistically – no**
- **Adjustable - No**

General structure

undefined

the description

The undefined property is a property of the global object, that is, it is a variable in the public domain, and the primary value of the undefined property is the undefined initial data type.

The undefined property is not configurable and is not writable in modern browsers (starting with the ECMAScript 5 standard), but try to avoid overwriting it.

Variables that have not been assigned a value will have an undefined value and the expressions will be undefined if a value is not assigned to the value variable. A function will return the undefined value if we do not specify a value to return that function via the return expression.

It is true that we can use the word undefined as an identifier in any field other than the public domain (because undefined is not a reserved word), but it is not appropriate to do so, and this will make it difficult to maintain and debug program errors.

```
// Do not do this at all
// "foo string"
(function () {var undefined = 'foo';
console.log (undefined, typeof undefined);}) ();

// "foo string"
(function (undefined)
{
console.log (undefined, typeof undefined);
}
)
('foo');
```

Examples

Coefficient of matching with undefined

You can use the coefficient of matching (===) with undefined to see if a variable has a value. In the following code, the value of variable x is not known, and the result of the if expression is true:

```
var x;
if (x === undefined) {
  // The software expressions will be implemented here
}
else {
  // The software expressions will not be implemented here
}
```

Note that we used the coefficient of matching (===) instead of using the normal equalizer (==), because x == undefined will return true if x is null, while the corresponding coefficient is not null. See the Comparison Transactions page for details.

The typeof parameter is the undefined value

Alternatively, we can use the typeof parameter to see if the value of the variable is undefined:

```
var x;
if (typeof x === 'undefined') {
  // The software expressions will be implemented here
}
```

One reason for using the typeof parameter is that it will not throw an error when the variable is not declared:

```
// The variable is not known yet
if (typeof x === 'undefined') {   // without problems
```

```
  // The software expressions will be implemented here
}

if (x === undefined) {   // Directs ReferenceError

}
```

The coefficient is void and the value is undefined

A third way to check if the value of an undefined variable is to use the void parameter:

```
var x;
if (x === void 0) {
  // The software expressions will be implemented here
}

// This variable has not been known before
if (y === void 0) {
  // The error will be called a - Uncaught ReferenceError: y is not defined
}
```

Null in JavaScript

Null represents the intentional absence of object value, which is an initial value type in JavaScript.

General structure

null

the description

Null is used via null, note that null is not an identifier of a property in the public object (ie it is not like the undefined property), but null indicates that there is no definition from the baseline, and indicates that the variable does not refer to any object.

```
// The variable does not exist, as it has not been known or created before
console.log (foo); // ReferenceError: foo is not defined

// The variable is present but has no type and no value
var foo = null;
console.log (foo); // null
```

The difference between null and undefined

When checking for null or undefined, consider the difference between the coefficient of equality (==) and the coefficient of matching (===). The first

factor will convert the species before comparison:

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

Word processing

These objects represent text strings and the ways they are processed and modified.

- String
- RegExp

The String object in JavaScript

The String object is a syntax function for text strings, or a set of characters.

General structure

The initial values of text strings can take the following form:

'string text'

"string text"

" 中文 español deutsch English हिन्दी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ ਪੰਜਾਬੀ 국어 மழழழ் Hebrew"

Text strings can be created using the String public object directly:

String (thing)

thing

Any value we want to convert to a text string.

Templates

Starting with the ECMAScript 2015, template literals can be created:

`hello world`

`hello!
 world! `

`hello $ {who}`

escape `<a> $ {who} </a>`

Smuggling of characters

In addition to the regular characters, special characters can be included by smuggling them as follows:

Code / output

\ 0 ** The character is NULL

\ ' ** Single quotation mark

\ " ** double quotation mark

\\ ** backslash

\ N ** New lineface (new line)

\ R ** Return carriage to the beginning of the line (carriage return)

\ V ** Vertical tab space

\ t **Scheduling distance (tab)

\ B ** Backspace

\ F ** Move to a new page (form feed)

\ UXXXX ** Unicode codepoint format

\ U {X} ... \ u {XXXXXX} ** Unicode codepoint format

\ XXX ** Latin-1 (Latin-1)

In contrast to some other programming languages, JavaScript does not differentiate between text strings enclosed in single quotation marks and text strings surrounded by double quotation marks; therefore, the previous smuggling characters will work regardless of the type of quotation marks around them.

Long text strings

Sometimes the code will contain some very long text strings, and instead of the long line or the line wrap in the editor, we can divide these text strings into several lines of code without affecting the content of the text strings itself. There are two ways to do this.

We can use the + operator to add text strings to each other as follows:

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

You can also use the backslash at the end of each line to indicate that the text string will be completed in the next line; but make sure that there is no space or any other character after the backslash (except for the new lineface), otherwise it will not work. Here's an example of this method:

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

Both previous examples will produce the same text string.

the description

Text strings are useful for holding data that represents textual words. One of the most commonly used operations on text strings is to check their length (across the length property), to collect text strings using + and +, and to check for or place subtext strings using the indexOf () function, The substring () function.

Access the characters

There are two ways to access the characters in a text string. The first is to use charAt ():

```
return 'cat'.charAt (1); // "a"
```

Another method added to the ECMAScript 5.1 standard is to treat the text string as matrix-like objects. The characters are associated with a numeric index:

```
return 'cat' [1]; // "a"
```

When you access characters using square brackets, trying to delete or assign a value to those attributes will not work; these attributes will not be writable or set

Compare text strings

String developers use strcmp () to compare text strings; in JavaScript, you can use operators larger than and smaller than:

```
var a = 'a';
var b = 'b';
if (a <b) {  // true
  console.log (a + 'is less than' + b);
} else if (a> b) {
  console.log (a + 'is greater than' + b);
} else {
  console.log (a + 'and' + b + 'are equal.');
}
```

A result similar to the above can be obtained by using the localeCompare () function that is available in the String object copies.

Differences between primary text strings and String objects

Note that JavaScript language distinguishes between String objects and initial text strings (the same applies to Boolean and Number objects).

Text strings defined by double or single quotation marks and text strings returned from the String function without the use of the new reserved word are primitive strings, and JavaScript automatically converts these initial values to String objects, allowing When using the function (or trying to access a property) on an initial string, JavaScript encapsulates the initial text string within an object and calls the respective function (or required property):

```
var s_prim = 'foo';
var s_obj = new String (s_prim);

console.log (typeof s_prim); // "string"
console.log (typeof s_obj); // "object"
```

Note that String objects and initial text strings produce different results when used with the eval () function, as the passed string of text will be treated as eval, and String objects will behave like other objects, ie, the object will be returned. The following example illustrates this:

```
var s1 = '2 + 2'; // Create a text string
var s2 = new String ('2 + 2'); // Create an object
console.log (eval (s1)); // Figure 4
console.log (eval (s2)); // Text string 2 + 2
```

For these reasons, codes can not work when you encounter String objects instead of primary text strings, but developers do not throw away these differences. We can convert the String object to the initial text string that is equivalent by using the valueOf () function:

```
console.log (eval (s2.valueOf ())); // Figure 4
```

Functions of the String function

- String.prototype

This property allows adding attributes that are available to all objects whose type is String.

Functions of the String function

*** String.fromCharCode ()**

Create a text string using a string of Unicode values.

String.fromCodePoint () *

Create a text string using a string of character values (code points).

*** String.raw ()**

Create a text string from a raw template.

- String.fromCharCode **()**

The String.fromCharCode () static function returns a constructed text string

using a string of Unicode character values.

General structure

String.fromCharCode (num1 [, ... [, numN]])

num1, ..., numN

A string of numbers representing Unicode values. The allowed domain is between 0 and 65535 (ie 0xFFFF), and the larger numbers of 0xFFFF will be truncated.

Returned value

A string containing characters that are linked to the Unicode values passed to the function.

the description

This function returns a primitive text string and does not return a String object.

Since the fromCharCode function is a static method of the String object, you should always use it as String.fromCharCode (), since you can not use it as part of your String object.

Examples

The following example shows the use of theCharCode () function:

```
String.fromCharCode (65, 66, 67); // "ABC"
String.fromCharCode (0x2014) // "-"
String.fromCharCode (0x12014) // "-" The character 1 will be ignored
```

Handle values larger than the maximum

It is true that Unicode values are commonly represented by a 16-bit character (as previously expected during the JavaScript drafting process). We could use the fromCharCode () function to recreate a single character from the majority of values (UCS-2 values, which are a subset of UTF 16 - where the most common characters).

For all valid Unicode values (up to 21 bits), using the fromCharCode () function alone is not appropriate, because characters with large values use two digits to represent a single character; therefore, String.fromCodePoint () Part of the ECMAScript 2015 standard (ES6) to return such pairs, allowing the representation of characters with values greater than the maximum.

Supports all browsers

- ## String.fromCodePoint ()

The String.fromCodePoint () static function returns a constructed text string using a string of code points.

General structure

String.fromCodePoint (num1 [, ... [, numN]])

num1, ..., numN

A string of numbers that represent the values of the characters (code points).

Returned value

A text string that contains characters associated with the values of the characters passed to the function.

Exceptions

The RangeError exception will be called if the invalid Unicode character value is passed (for example: "RangeError: NaN is not a valid code point").

the description

This function returns a primitive text string and does not return a String object.

Since the fromCodePoint function is a static method of the String object, you should always use it as String.fromCodePoint (), since you can not use it as part of your String object.

Examples

The following examples show the use of theCodePoint () function:

String.fromCodePoint (42); "" * "

String.fromCodePoint (65, 90); // "AZ"

String.fromCodePoint (0x404); // "\ u0404"

String.fromCodePoint (0x2F804); // "\ uD87E \ uDC04"

String.fromCodePoint (194564); // "\ uD87E \ uDC04"

String.fromCodePoint (0x1D306, 0x61, 0x1D307) // "\ uD834 \ uDF06a \ uD834 \ uDF07"

String.fromCodePoint ('_'); // RangeError

String.fromCodePoint (Infinity); // RangeError

String.fromCodePoint (-1); // RangeError

String.fromCodePoint (3.14); // RangeError

String.fromCodePoint (3e-2); // RangeError

String.fromCodePoint (NaN); // RangeError

The String.fromCharCode () function can not find characters whose values are greater than the maximum. The following example will return a 4-byte character, plus 2-byte characters (ie, it returns one character that represents a 2-bit text string instead Of 1):

console.log (String.fromCodePoint (0x2F804));

Reduced support for browsers

The String.fromCodePoint function has been added to ECMAScript 2015 (ES6) and is not yet supported in all browsers, so the following code can be

used to identify it:

(This code for professionals / skip it and return to it after the book is finished)

```
/ *! http://mths.be/fromcodpoint v0.1.0 by @mathias * /
if (! String.fromCodePoint) {
  (function () {
    var defineProperty = (function () {
      // IE 8 only supports `Object.defineProperty` on DOM elements
      try {
        var object = {};
        var $ defineProperty = Object.defineProperty;
        var result = $ defineProperty (object, object, object) && $
defineProperty;
      } catch (error) {}
      return result;
    } ());
    var stringFromCharCode = String.fromCharCode;
    var floor = Math.floor;
    var fromCodePoint = function () {
      var MAX_SIZE = 0x4000;
      var codeUnits = [];
      var highSurrogate;
      var lowSurrogate;
      var index = -1;
      var length = arguments.length;
      if (! length) {
        return '';
      }
```

```
	var result = '';
	while (++ index <length) {
	  var codePoint = Number (arguments [index]);
	  if (
	    ! isFinite (codePoint) || // `NaN`,` + Infinity`, or `-Infinity`
	    codePoint <0 || // not a valid Unicode code point
	    codePoint> 0x10FFFF || // not a valid Unicode code point
	    floor (codePoint)! = codePoint // not an integer
	  ) {
	    throw RangeError ('Invalid code point:' + codePoint);
	  }
	  if (codePoint <= 0xFFFF) {  // BMP code point
	    codeUnits.push (codePoint);
	  } else {  // Astral code point; split in surrogate halves
	    // http://mathiasbynens.be/notes/javascript-encoding#surrogate-formulae
	    codePoint - = 0x10000;
	    highSurrogate = (codePoint >> 10) + 0xD800;
	    lowSurrogate = (codePoint% 0x400) + 0xDC00;
	    codeUnits.push (highSurrogate, lowSurrogate);
	  }
	  if (index + 1 == length || codeUnits.length> MAX_SIZE) {
	    result + = stringFromCharCode.apply (null, codeUnits);
	    codeUnits.length = 0;
	  }
	}
	return result;
```

```
    };
    if (defineProperty) {
      defineProperty (String, 'fromCodePoint', {
        'value': fromCodePoint,
        'configurable': true,
        'writable': true
      });
    } else {
      String.fromCodePoint = fromCodePoint;
    }
  } ());
}
```

Support browsers

// Chrome // Firefox // Internet Explorer // Opera // Safari
// 41 // 29 // not supported //28 // 10

String.raw ()

The String.raw () function is a tag function for template literals, similar to the preceding r in Python or the previous @ in C # (but there are differences between them). Used to get the raw text string from the template (ie, unexplained text).

General structure

String.raw (callSite, ... substitutions)

String.raw`templateString`

callSite

Object to call template like: {raw: ['foo', 'bar', 'baz']}.

... substitutions

The values to be replaced.

templateString

A text string representing the template, and the values to be replaced ($ {...}).

Returned value

A raw text string for the text string that represents the template.

Exceptions

The TypeError exception will be called if the first broker is not a properly defined object.

the description

In most cases, the String.raw () function is used with text strings that represent the templates. The first format mentioned above is rarely used, because the JavaScript engine will automatically call this format with the necessary arguments.

Examples

The following examples show that the raw () function is used:

```
String.raw`Hi \ n $ {2 + 3}! `;
// 'Hi \ n5!', The character after 'Hi'
// is not a newline character,
// '\' and 'n' are two characters.

String.raw`Hi \ u000A! `;
// 'Hi \ u000A!', Same here, this time we will get the
// \, u, 0, 0, 0, A, 6 characters.
// All kinds of escape characters will be ineffective
// and backslashes will be present in the output string.
// You can confirm this by checking the .length property
// of the string.
```

```
let name = 'Bob';
String.raw`Hi \ n $ {name}! `;
// 'Hi \ nBob!', Substitutions are processed.

// Normally you would not call String.raw () as a function,
// but to simulate `t $ {0} e $ {1} s $ {2} t` you can do:
String.raw ({raw: 'test'}, 0, 1, 2); // 't0e1s2t'
// Note that 'test', a string, is an array-like object
// The following is equivalent to
// `foo $ {2 + 3} bar $ {'Java' + 'Script'} baz`
String.raw ({
  raw: ['foo', 'bar', 'baz']
}, 2 + 3, 'Java' + 'Script'); // 'foo5barJavaScriptbaz'
```