



ROLLING CODE
SCHOOL

Módulo 2 - Full Stack

Introducción	6
El Software y sus características	8
Estructura Interna de una Computadora	10
Redes de Computadoras	13
Programación y construcción de Software	16
Los sistemas y su enfoque	17
¿Qué es un Sistema?	17
Características de los sistemas	18
Intercambio entre sistemas	20
Sistemas tecnológicos	21
¿Cómo se construye el Software?	23
Fundamentos de programación	25
Diseño de Algoritmos	25
Algoritmos	27
Características de los algoritmos	27
Herramientas para la representación gráfica de los algoritmos	30
Diagramas de Flujo	30
Pseudocódigo	31
Lenguajes de Programación	32

Tipos de Lenguajes de Programación	40
Nivel de Abstracción del Procesador	41
Paradigma de Programación	41
Tipos de Lenguajes de Programación	44
¿Qué es un Programa?	46
Estructuras de control	56
Recursividad	66
Estructuras de Datos: Pilas, Colas y Listas	68
Javascript	71
¿Qué es JavaScript?	71
¿Qué debo conocer previamente?	71
Historia	72
JavaScript y Java	74
ECMAScript	75
Ejecutando código JavaScript en el navegador	75
Comentarios	79
Tipos y operadores	80
Cadenas de texto (Strings)	81
Imprimir una cadena de texto	82
Concatenar cadenas	82
Números (Number)	83
Valores y expresiones booleanas (Boolean)	84
Variables	86
Declaraciones	87
Creando variables	87
Diferencia entre var y let	88
La utilidad de las variables	88
Reasignar o cambiar el valor de las variables	89
Variables sin valor o undefined	90
Variables con null	91
Constantes	91
¿Dónde y cuánto vive una variable?	92
Condicionales o estructuras de control	93
Condicionales simples	93
Condicionales dobles o De lo contrario (else)	95
Condiciones múltiples - anidados	96
Condicionales múltiples - De lo contrario, si (else if)	96
Condicionales múltiples - según sea , el caso de	97
Condiciones compuestas	99
Forma de pensar como un programador	100

Evaluación de expresiones booleanas	101
Estructuras repetitivas - Ciclos	105
Ciclo While (Mientras)	105
Ciclo Do While (Hacer Mientras)	107
Ciclo for	108
Ejemplos con while y for	110
Arreglos	111
Obtener elementos del arreglo	112
Recorriendo un arreglo	112
Reemplazar un elemento	113
Agregar nuevos elementos	113
Eliminar elementos	114
Funciones	114
Argumentos o Parámetros	115
Retornar un valor	117
Estructura de una función	118
Cajas negras	118
Ejemplo	119
Objetos	119
Obtener valores de un objeto	120
Aregar nuevas propiedades al objeto	121
Modificar propiedades del objeto	121
Eliminar propiedades de un objeto	122
Recorrer las propiedades de un objeto	122
Formas de crear objetos	123
Con notación literal	124
Con función constructora	124
El constructor Object()	124
Object.create	125
Comparando Objetos	125
Los objetos son de tipo referencia en JavaScript. Dos objetos con las mismas propiedades y métodos nunca son iguales. Sólo comparando la misma referencia al objeto consigo mismo dará como resultado true.	125
Objetos globales	126
Object	126
Array	127
Push	128
Pop	128
Sort	129
Join	130
Slice	130

Splice	131
Function	131
Boolean	133
Number	134
String	134
Math	136
Date	137
Programación Orientada a Objetos (POO)	139
Elementos de la POO	140
Clase	140
Definiendo clases usando ES6	141
Objetos e instancias	141
Constructor	142
Propiedades	143
Métodos	144
Definiendo métodos en ES6	144
Settes y Getters	145
Propiedades computadas de objetos JavaScript	145
Herencia	147
Prototipos	147
Prototype Chain(Cadena de prototipos)	148
Moviendo el método al prototipo	150
Prototipos y clases	151
Aplicando herencia	152
Aplicando herencia usando ES6	153
Encapsulación	155
Abstracción	155
Polimorfismo	155
Javasctipt en el navegador	157
Detectar funcionalidades	157
BOM	158
Propiedades de window	158
window.navigator	158
window.location	158
window.history	160
window.frames	160
window.screen	160
Métodos de window	161
window.open(), window.close()	161
window.moveTo(), window.moveBy(), window.resizeTo(), window.resizeBy()	161

window.alert(), window.prompt(), window.confirm()	161
window.setTimeout(), window.setInterval()	162
El objeto document	164
DOM	164
Accediendo a los nodos	165
Acceder al contenido de un tag	167
Acceso directo a tags	168
Parent y Childs	168
Modificando los nodos	169
Creando y eliminando nodos	170
Objetos DOM sólo de HTML	172
Eventos	173
Capturar eventos	174
Modelo tradicional	174
Modelo avanzado	175
Eliminar manejadores de eventos	176
Detener el flujo de eventos	176
Delegación de eventos	177
Listado de manejadores de eventos	177
Métodos de evento disponibles en JavaScript	181
Eventos onLoad y onUnLoad	182
JSON	182
JSON.stringify	184
JSON.parse	185
Web Storage	186
Características de Local Storage y Session Storage:	186
Local Storage	186
¿Qué es localStorage?	186
Ajax	190
Arquitectura Cliente Servidor	191
Componentes cliente servidor	192
Ajax introducción	192
Modelo de aplicación web clásico (Síncrono)	193
Modelo de aplicación web con ajax (Asíncrono)	193
Qué significa AJAX	194
Iniciando una petición o request AJAX	194
Procesando la respuesta	195
Ejemplo completo	196

Introducción

Cada uno de los tres últimos siglos ha estado dominado por una **nueva tecnología**. El siglo xviii fue la época de los grandes sistemas mecánicos que dieron paso a la Revolución Industrial. El siglo xix fue la era de la máquina de vapor. Durante el siglo xx, la **tecnología clave fue la recopilación, procesamiento y distribución de información**. Entre otros desarrollos vimos la instalación de las redes telefónicas a nivel mundial, la invención de la radio y la televisión, el nacimiento y crecimiento sin precedentes de la industria de la computación, el lanzamiento de satélites de comunicaciones y, desde luego, Internet. En 1977 Ken Olsen era presidente de Digital Equipment Corporation, en ese entonces la segunda empresa distribuidora de computadoras más importante del mundo (después de IBM). Cuando se le preguntó por qué Digital no iba a incursionar a lo grande en el mercado de las computadoras personales, dijo: "**No hay motivos para que una persona tenga una computadora en su hogar**". La historia demostró lo contrario y Digital desapareció. En un principio, las personas compraban computadoras para el procesamiento de palabras y para juegos. En los últimos años, probablemente la razón más importante sea acceder a Internet. En la actualidad disponemos de un abanico de dispositivos tecnológicos que facilitan las tareas cotidianas, desde electrodomésticos que simplifican las tareas del hogar hasta computadoras, notebooks, smartphones y tablets que nos permiten trabajar, comunicarnos a través de Internet, esparcirnos y mucho más. Cada uno de ellos ha pasado por un proceso de ideación, planificación, desarrollo, manufactura y logística que permitieron que el producto terminado esté disponible para nosotros. En 1965, **Gordon Moore** (Cofundador de Intel) afirmó que la tecnología tenía futuro, que el número de transistores en los circuitos integrados, uno de los componentes esenciales en el procesador de una computadora, se duplicaba cada año y que la tendencia continuaría durante las siguientes dos décadas. Aunque luego disminuyó este periodo a 2 años, esta ley empírica se ha cumplido y se traduce en que tengamos cada día dispositivos más pequeños, más veloces y a un costo más bajo. Tanto es así, que empresas de electrónica ya se encuentran desarrollando electrodomésticos con programas inteligentes que pueden conectarse a través de Internet, de forma tal de que podamos consultar el estado del trabajo de nuestro lavarropas desde la oficina o en viaje. A pesar de que la industria de la computación es joven si se la compara con otras (como la automotriz y la de transporte aéreo), las computadoras han progresado

de manera espectacular en un periodo muy corto. Durante las primeras dos décadas de su existencia, estos sistemas estaban altamente centralizados y por lo general se encontraban dentro de un salón grande e independiente. Era común que este salón tuviera paredes de vidrio, a través de las cuales los visitantes podían mirar boquiabiertos la gran maravilla electrónica que había en su interior. Una empresa o universidad de tamaño mediano apenas lograba tener una o dos computadoras, mientras que las instituciones muy grandes tenían, cuando mucho, unas cuantas docenas. La idea de que en un lapso de 40 años se produjeran en masa miles de millones de computadoras mucho más poderosas y del tamaño de una estampilla postal era en ese entonces mera ciencia ficción.

El Software y sus características

El software en sus comienzos era la parte insignificante del hardware, lo que venía como añadidura, casi como regalo. Al poco tiempo adquirió una entidad propia. En la actualidad, el software es la tecnología individual más importante en el mundo. Nadie en la década de 1950 podría haber predicho que el software se convertiría en una tecnología indispensable en los negocios, la ciencia, la ingeniería; tampoco podría preverse que una compañía de software podría volverse más grande e influyente que la mayoría de las compañías de la era industrial; que una red construida con software, llamada Internet cubriría y cambiaría todo, desde la investigación bibliográfica hasta las compras de los consumidores y los hábitos de las personas. Nadie podría haber imaginado que estaría relacionado con sistemas de todo tipo: transporte, medicina, militares, industriales, entretenimiento, automatización de hogares.

Una definición formal de software según la IEEE (Instituto de Ingeniería Eléctrica y Electrónica) es la siguiente:

Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación.

El software puede definirse como “el alma y cerebro de la computadora, la corporización de las funciones de un sistema, el conocimiento capturado acerca de un área de aplicación, la colección de los programas, y los datos necesarios para convertir a una computadora en una máquina de propósito especial diseñada para una aplicación particular, y toda la información producida durante el desarrollo de un producto de software”. El software viabiliza el producto más importante de nuestro tiempo: la información.

Características del software:

1. El software es intangible, es decir, que se trata de un concepto abstracto.
2. Tiene alto contenido intelectual.

-
- 3.** Su proceso de desarrollo es humano intensivo, es decir que la materia prima principal radica en la mente de quienes lo crean.
 - 4.** El software no exhibe una separación real entre investigación y producción.
 - 5.** El software puede ser potencialmente modificado, infinitamente.
 - 6.** El software no se desgasta
 - 7.** La mayoría del software, en su mayoría, aún se construye a medida.
 - 8.** El software no se desarrolla en forma masiva, debido a que es único

Estructura Interna de una Computadora

Una computadora moderna consta de uno o más procesadores, una memoria principal, discos, impresoras, un teclado, un ratón, una pantalla o monitor, interfaces de red y otros dispositivos de entrada/salida. En general es un sistema complejo. Si todos los programadores de aplicaciones tuvieran que comprender el funcionamiento de todas estas partes, no escribirían código alguno. Es más: el trabajo de administrar todos estos componentes y utilizarlos de manera óptima es una tarea muy desafiante. Por esta razón, las computadoras están equipadas con una capa de software llamada sistema operativo, cuyo trabajo es proporcionar a los programas de usuario un modelo de computadora mejor, más simple y pulcro, así como encargarse de la administración de todos los recursos antes mencionados.

La mayoría de las computadoras, grandes o pequeñas, están organizadas como se muestra en la siguiente figura. Constan fundamentalmente de **tres componentes principales**: Unidad Central de Proceso (UCP) o procesador, la memoria principal o central.

Si a los componentes anteriores se les añaden los dispositivos para comunicación con la computadora, aparece la estructura típica de un sistema de computadora: dispositivos de entrada, dispositivos de salida, memoria externa y el procesador/memoria central.

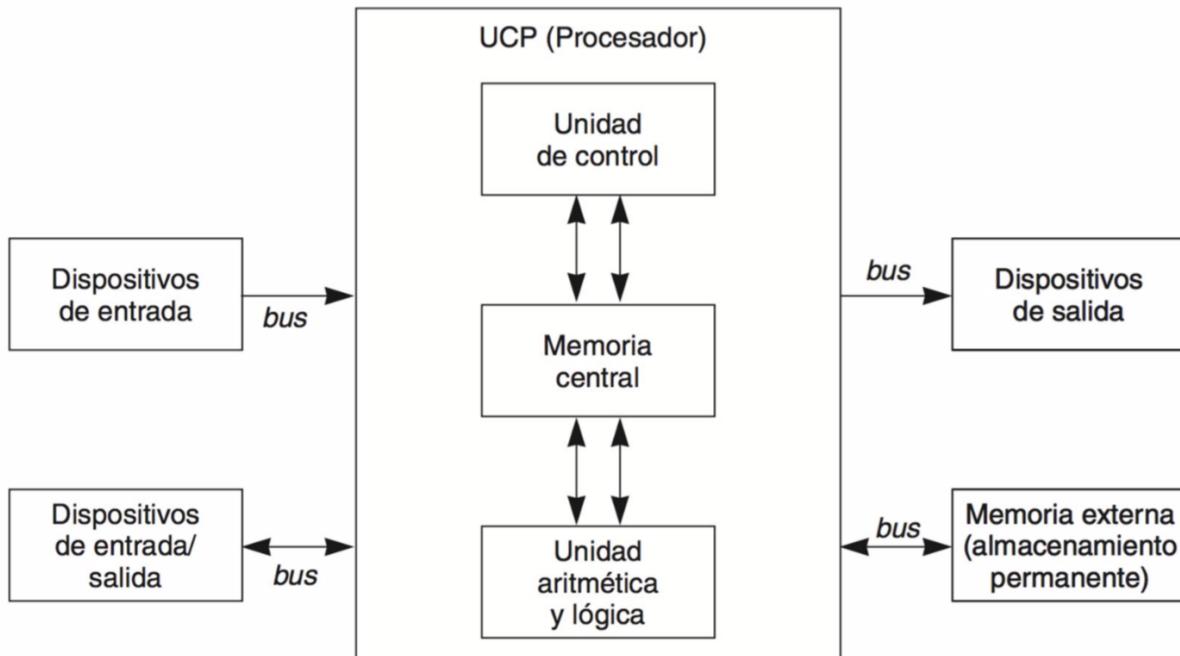


Figura 1: Estructura interna de un sistema de computadora

Los dispositivos de **Entrada/Salida (E/S)** (en inglés, Input/Output I/O) o periféricos permiten la comunicación entre la computadora y el usuario.

Los **dispositivos de entrada**, como su nombre indica, sirven para introducir datos en la computadora para su proceso. Los datos se leen de los dispositivos de entrada y se almacenan en la memoria central o interna. Los dispositivos de entrada convierten la información de entrada en señales eléctricas que se almacenan en la memoria central. Dispositivos de entrada típicos son teclados, lápices ópticos, joysticks, lectores de códigos de barras, escáneres, micrófonos, lectores de tarjetas digitales, lectores RFID (tarjetas de identificación por radio frecuencia), etc. Hoy, tal vez el dispositivo de entrada más popular es el ratón (mouse) que mueve un puntero gráfico (electrónico) sobre la pantalla, o más recientemente las pantallas táctiles, que facilitan la interacción usuario-máquina.

Los **dispositivos de salida** permiten representar los resultados (salida) del proceso. El dispositivo de salida típico es la pantalla o monitor. Otros dispositivos de salida son:

impresoras (imprimen resultados en papel), trazadores gráficos (plotters), reconocedores (sintetizadores) de voz, parlantes, entre otros.

Los dispositivos de entrada/salida y dispositivos de almacenamiento masivo o auxiliar (memoria externa) son: unidad de discos (disquetes, CD-ROM, DVD, discos duros, etc.), videocámaras, memorias flash, USB, etc.

La **memoria central o simplemente memoria** (interna o principal) se utiliza para almacenar información (RAM, del inglés Random Access Memory). En general, la información almacenada en la memoria puede ser de dos tipos: instrucciones de un programa y datos con los que operan las instrucciones. Por ejemplo, para que un programa se pueda ejecutar (correr, funcionar..., en inglés, run), debe ser situado en la memoria central, en una operación denominada carga (load) del programa. Después, cuando se ejecuta el programa, cualquier dato a procesar se debe llevar a la memoria mediante las instrucciones del programa. En la memoria central, hay también datos diversos y espacio de almacenamiento temporal que necesita el programa cuando se ejecuta a fin de poder funcionar.

La memoria central de una computadora es una zona de almacenamiento organizada en centenares o miles de unidades de almacenamiento individual o celdas. La memoria central consta de un conjunto de celdas de memoria (estas celdas o posiciones de memoria se denominan también palabras, aunque no guardan analogía con las palabras del lenguaje). El número de celdas de memoria de la memoria central, depende del tipo y modelo de computadora; hoy día el número suele ser millones (512, 1.024, etc.). Cada celda de memoria consta de un cierto número de bits (normalmente 8, un byte).

La unidad elemental de memoria se llama **byte**. Un byte tiene la capacidad de almacenar un carácter de información, y está formado por un conjunto de unidades más pequeñas de almacenamiento denominadas bits, que son dígitos binarios que pueden asumir como valor un 0 o un 1.

Siempre que se almacena una nueva información en una posición, se destruye (desaparece) cualquier información que en ella hubiera y no se puede recuperar. La dirección es permanente y única, el contenido puede cambiar mientras se ejecuta un programa.

La **memoria central de una computadora** puede tener desde unos centenares de miles de bytes hasta millones de bytes. Como el byte es una unidad elemental de almacenamiento, se utilizan múltiplos de potencia de 2 para definir el tamaño de la memoria central: Kilobyte (KB o Kb) igual a 1.024 bytes (2^{10}) –prácticamente se consideran 1.000–; Megabyte (MB o Mb) igual a 1.024×1.024 bytes = = 1.048.576 (2^{20}) –prácticamente se consideran 1.000.000; Gigabyte (GB o Gb) igual a 1.024 MB (2^{30}), 1.073.741.824 = prácticamente se consideran 1.000 millones de MB.

Byte	Byte (B)	<i>equivale a</i>	8 bits
Kilobyte	Kbyte (KB)	<i>equivale a</i>	1.024 bytes
Megabyte	Mbyte (MB)	<i>equivale a</i>	1.024 Kbytes
Gigabyte	Gbyte (GB)	<i>equivale a</i>	1.024 Mbytes
Terabyte	Tbyte (TB)	<i>equivale a</i>	1.024 Gbytes

$$1 \text{ Tb} = 1.024 \text{ Gb} = 1.024 \times 1.024 \text{ Mb} = 1.048.576 \text{ Kb} = 1.073.741.824 \text{ B}$$

Tabla 1: Unidades de medida para el almacenamiento en la memoria

La **Unidad Central de Proceso UCP**, o procesador, dirige y controla el proceso de información realizado por la computadora. La UCP procesa o manipula la información almacenada en memoria; puede recuperar información desde memoria (esta información son datos o instrucciones de programas) y también puede almacenar los resultados de estos procesos en memoria para su uso posterior.

Redes de Computadoras

La fusión de las computadoras y las comunicaciones ha tenido una profunda influencia en cuanto a la manera en que se organizan los sistemas de cómputo. El concepto una vez dominante del “centro de cómputo” como un salón con una gran computadora a la que los usuarios llevaban su trabajo para procesarlo es ahora totalmente obsoleto, (aunque los centros de datos que contienen miles de servidores de Internet se están volviendo comunes). El viejo modelo de una sola computadora para atender todas las necesidades computacionales de la organización se ha reemplazado por uno en el que un gran número

de computadoras separadas pero interconectadas realizan el trabajo. A estos sistemas se les conoce como redes de computadoras.

Se dice que dos computadoras están interconectadas si pueden intercambiar información. La conexión no necesita ser a través de un cable de cobre; también se puede utilizar fibra óptica, microondas, infrarrojos y satélites de comunicaciones. Las redes pueden ser de muchos tamaños, figuras y formas, como veremos más adelante. Por lo general se conectan entre sí para formar redes más grandes, en donde Internet es el ejemplo más popular de una red de redes.

Imaginemos el sistema de información de una empresa como si estuviera constituido por una o más bases de datos con información de la empresa y cierto número de empleados que necesitan acceder a esos datos en forma remota. En este modelo, los datos se almacenan en poderosas computadoras denominadas servidores. A menudo estos servidores están alojados en una ubicación central y un administrador de sistemas se encarga de su mantenimiento. Por el contrario, los empleados tienen en sus escritorios máquinas más simples conocidas como clientes, con las cuales acceden a los datos remotos, por ejemplo, para incluirlos en las hojas de cálculo que desarrollan (algunas veces nos referiremos al usuario humano del equipo cliente como el "cliente", aunque el contexto debe dejar en claro si nos referimos a la computadora o a su usuario). Las máquinas cliente y servidor se conectan mediante una red, como se muestra en la figura 2.

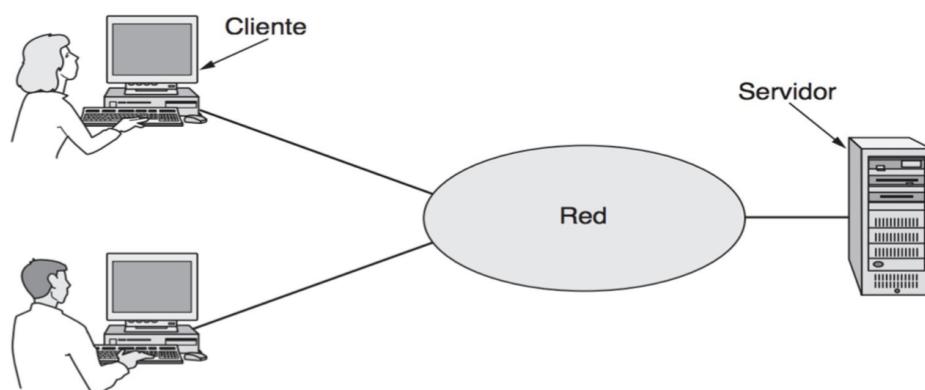


Figura 2: Esquema de una red de computadoras

A esta disposición se le conoce como **modelo cliente-servidor**. Es un modelo ampliamente utilizado y forma la base de muchas redes. La realización más popular es la de una aplicación web, en la cual el servidor genera páginas web basadas en su base de datos en respuesta a las solicitudes de los clientes que pueden actualizarla. El modelo cliente-servidor es aplicable cuando el cliente y el servidor se encuentran en el mismo edificio (y pertenecen a la misma empresa), pero también cuando están muy alejados. Por ejemplo, cuando una persona accede desde su hogar a una página en Internet se emplea el mismo modelo, en donde el servidor web remoto representa al servidor y la computadora personal del usuario representa al cliente. En la mayoría de las situaciones un servidor puede atender un gran número (cientos o miles) de clientes simultáneamente.

La evolución de las comunicaciones y los dispositivos personales, así como las necesidades emergentes de compartir información en tiempo real han posibilitado la expansión de Internet a todos los rincones del mundo. De esta forma cualquier persona puede acceder a sus archivos, compartir datos, comunicarse o buscar información en cualquier momento a través de su computadora, notebook, teléfonos celulares entre otros, tal como se muestra en la siguiente figura.



Figura 3: Integración de tecnología a través de Internet

Programación y construcción de Software

El único tipo de instrucciones que una computadora puede entender es el ***lenguaje de máquina***, o lenguaje de bajo nivel, donde diferentes tipos de procesadores pueden tener distintos lenguajes de máquina. El lenguaje máquina está compuesto de ceros y unos lo que hace que programar en lenguaje máquina sea un proceso tedioso y sujeto a errores.

Una alternativa a utilizar lenguaje de máquina es el lenguaje Assembly, Assembler o ensamblador, que es también un lenguaje de bajo nivel que utiliza mnemonics (o abreviaturas) y es más fácil de entender que ceros y unos. Sin embargo, el único lenguaje que una computadora puede entender directamente es el lenguaje máquina, ¿entonces cómo es posible que entienda lenguajes como Assembler? La respuesta es que el lenguaje Assembler es convertido o traducido a lenguaje de máquina mediante un programa llamado ensamblador. Es importante destacar que hay una correspondencia directa entre el lenguaje Assembler y el lenguaje máquina, lo que significa que para cada instrucción de lenguaje assembler existe una instrucción de lenguaje máquina, lo que hace la traducción un proceso directo.

Sin embargo, más allá de que el lenguaje Assembler es más sencillo que el lenguaje máquina, distintos tipos de procesadores tienen diferentes conjuntos de instrucciones lo que se traduce en distintos dialectos de Assembler de una computadora a otra.

La solución para hacer la tarea de programación más sencilla y posibilitar a los programas funcionar en distintos tipos de computadoras es utilizar lenguajes de alto nivel, que son más similares al lenguaje natural que utilizamos para comunicarnos diariamente y por motivos históricos estos lenguajes utilizan palabras del idioma inglés. Uno de los primeros lenguajes de programación de alto nivel fue FORTRAN (del inglés FORmula TRANslation, o traducción de fórmulas) que fue desarrollado en los comienzos de los años 50 para ayudar a resolver problemas matemáticos. Desde ese entonces, una gran cantidad de lenguajes de programación de alto nivel han sido creados para abordar distintos tipos de problemas y solucionar las necesidades de distintos tipos de usuarios. Algunos de ellos incluyen a COBOL, también desarrollado en los 50 para abordar aplicaciones empresariales y de negocios; BASIC en los 60 para programadores recién iniciados, Pascal en los 70 para problemas científicos, C, C++ y muchos otros. En este material nos centraremos en el

lenguaje Javascript, también de alto nivel y de propósito general: es decir que puede usarse para una gran variedad de problemas y rubros.

Los sistemas y su enfoque

¿Por qué hablamos de sistemas?

En la primera mitad del siglo XX, surgió la necesidad de diseñar métodos de investigación y estudio de los fenómenos complejos a causa de una acumulación de problemáticas en las que los métodos de investigación de las ciencias particulares se mostraban insuficientes. Por un lado, los nuevos sistemas de producción que incluían varias automatizaciones, el manejo de grandes cantidades de energía (termoeléctrica, nuclear...) que requería de especialistas de variadas ramas, el desarrollo y organización de transporte terrestre, marítimo y aéreo y otros fenómenos. Por otro, los grandes desarrollos científicos en la física (relatividad, estructura atómica, mecánica cuántica), biología (genética, evolución, estudio de poblaciones), química (teoría del enlace de Lewis, tabla periódica, estructura cristalina), matemática (álgebra de Boole, desarrollo del cálculo, problemas de Hilbert). Estas grandes revoluciones en el hacer y el pensar hicieron necesario el desarrollo de un enfoque complejo para la investigación de fenómenos complejos. Así nació el enfoque sistémico, sustentado por la Teoría General de los Sistemas (TGS) formulada por Ludwig von Bertalanffy a mediados del siglo XX.

Bertalanffy se dedicó especialmente a los organismos como sistemas biológicos, pero luego generalizó su estudio a todo tipo de sistemas. De tal manera que hoy se utiliza el término sistema en todas las áreas del conocimiento humano.

¿Qué es un Sistema?

Llamamos sistema a todo conjunto de elementos relacionados entre sí –puede ser por una finalidad en común, que tienen un cierto orden u organización y que cumplen una función.

Los sistemas tienen composición (los elementos que lo forman), una estructura interna dada por el conjunto de relaciones entre sus componentes. Y también tienen un entorno o

ambiente que es el conjunto de cosas que no pertenecen al sistema pero que actúan sobre él o sobre las que él actúa intercambiando materia, energía e información (MEI).

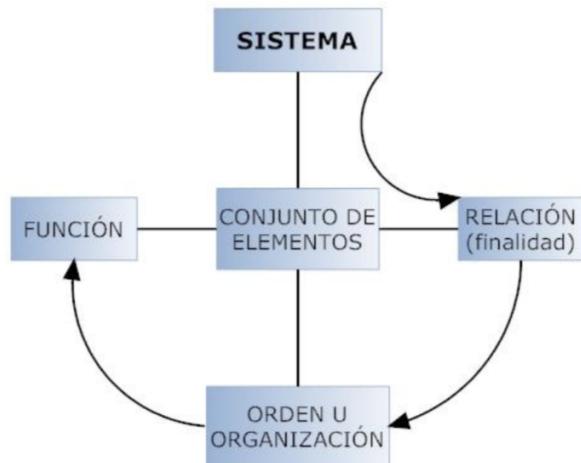


Figura 4: Elementos de un sistema

Los sistemas están inmersos en un entorno o ambiente, que es el conjunto de elementos que está fuera del sistema, es decir que no pertenecen al sistema pero que actúan sobre él o sobre las que el sistema actúa intercambiando materia, energía e información (MEI).

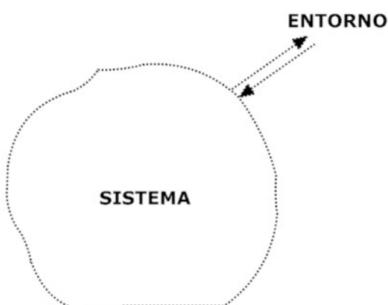


Figura 5: Relación del sistema con su entorno

Características de los sistemas

La **característica principal de los sistemas** es que poseen una propiedad emergente que no poseen sus componentes particulares. Por ejemplo, la vida es la propiedad emergente de un sistema compuesto por huesos, órganos, etc.; marchar es la propiedad emergente del

sistema automóvil compuesto por chapas, motor, luces, etc. Este hecho se suele enunciar con la siguiente afirmación

EL TODO ES MÁS QUE LA SUMA DE LAS PARTES

Otras características de los sistemas son:

1. **Límite o frontera:** Son demarcaciones que permiten establecer qué elementos pertenecen o no al sistema. Los límites pueden ser:
 - Concretos: los que tienen existencia material (ríos que separan países, paredes que definen aulas, etc.)
 - Simbólicos: los que no tienen existencia material y vienen dados por acuerdos, reglas o normas (un alumno pertenece a un curso porque lo establece la escuela, más allá de que pueda hallarse en otro salón o fuera de la misma)
2. **Depósitos o almacenamientos:** son lugares donde se almacena materia, energía o información (MEI). Los depósitos pueden ser:
 - Permanentes: aquellos en que están diseñados para que su contenido no se altere (CD- ROM, libros, carteles fijos, etc.)
 - Transitorios: aquellos diseñados para que su contenido sufra modificaciones (pizarrón, cartuchera, tanques de agua, etc.)
3. **Canales:** Son lugares o conductos por donde circula materia, energía o información (MEI). Los canales pueden comunicar dos sistemas entre sí o partes de un mismo sistema (las calles pueden ser canales de materia, los cables pueden ser canales de energía si llevan corriente o de información si son telefónicos o de redes, etc.)
4. **Subsistemas:** los sistemas complejos (muchos componentes y relaciones entre ellos) pueden dividirse para su estudio en subsistemas. Esto permite diferentes niveles de estudio de los mismos. Se llama nivel cero al análisis del sistema en su totalidad y su intercambio con el entorno. A partir de allí se define el nivel 1, nivel 2,

etc.

Niveles y subsistemas del automóvil.

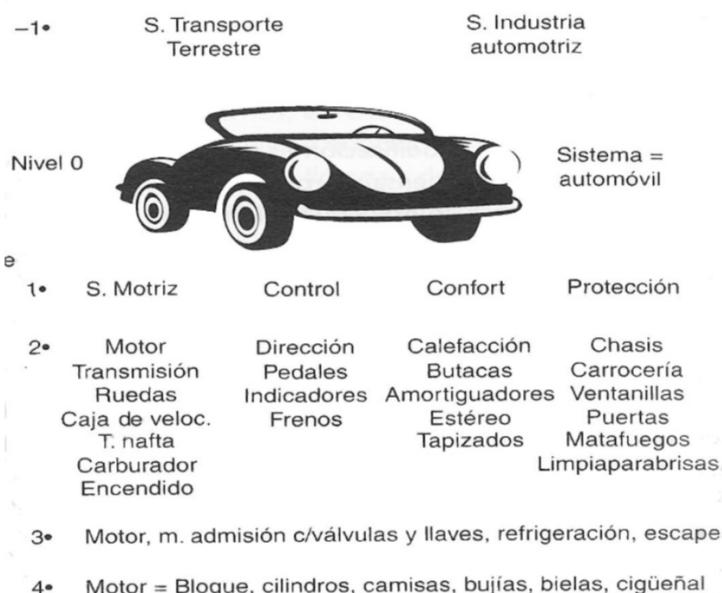


Figura 6: Sistemas y Subsistemas

Intercambio entre sistemas

Los sistemas intercambian entre sí materia, energía e información (MEI). Para que se dé este intercambio es necesario que MEI atraviese los límites del sistema hacia (o desde) el entorno. Si el sistema intercambia con el medio se dice que es abierto, de lo contrario se considera cerrado.

En **sistemas cerrados** cualquier estado final está determinado por sus condiciones iniciales, ya que no hay modo de que el entorno actúe sobre él. Si un sistema cerrado tampoco intercambia energía se dice que es aislado. En realidad, el único sistema que se considera absolutamente aislado es el universo. De igual modo, muchos sistemas mecánicos e informáticos pueden considerarse razonablemente cerrados.

Los **sistemas abiertos**, en cambio, pueden, crecer, cambiar, adaptarse al ambiente, incluso algunos reproducirse. Si un sistema posee la organización necesaria para controlar su propio desarrollo, asegurando la continuidad de su composición y estructura (homeostasis) y la de los flujos y transformaciones con que funciona (homeorresis) –mientras las perturbaciones producidas desde su entorno no superen cierto grado–, entonces el sistema es autopoyético. Los seres vivos, los ecosistemas y organizaciones sociales pueden considerarse sistemas abiertos.

Estos flujos de MEI se pueden representar en diagramas como el siguiente



Figura 7: Entradas y Salidas

Para clarificar, las líneas de los diferentes flujos pueden representarse por diferentes colores o trazos.

Este es el nivel cero de representación de un sistema, con las entradas y salidas de MEI que atraviesan sus límites. Este tipo de representaciones se denomina diagrama de entrada y salida (E/S o U/O) o diagrama de caja negra, ya que no interesa mostrar qué sucede dentro del sistema.

Sistemas tecnológicos

Los sistemas tecnológicos, son aquellos diseñados por los seres humanos para que cumplan con una finalidad específica. Por eso se dice que son sistemas teleológicos artificiales (del griego telos = fin). La orientación para al fin que se busca suele definir la propiedad emergente del sistema tecnológico. En el ejemplo del automóvil, la propiedad emergente de marchar también se busca como finalidad o propósito del sistema.

Es conveniente aclarar que los sistemas son recortes de la realidad que alguien se propone estudiar o considerar; a ese recorte se le llama Abstracción. En algunos sistemas tecnológicos como un automóvil es sencillo identificar este recorte. Sin embargo, en la red de generación y distribución de energía eléctrica del país no resulta tan sencillo.

Algunos sistemas tecnológicos se caracterizan por procesar materia: son los sistemas de procesamiento de materia (SM). Estos están diseñados para producir, procesar, generar, transformar o distribuir materiales. Las industrias, las huertas, las licuadoras, etc. pueden considerarse SM.

Otros se caracterizan por procesar energía, los sistemas de procesamiento de energía (SE). Estos están diseñados para generar, transformar, distribuir energía. Los ventiladores, automóviles, represas hidroeléctricas, explosivos, etc. pueden considerarse SE.

Los que se caracterizan por procesar información se llaman sistemas de información (SI). Están diseñados con el fin de generar, transformar y distribuir información entre otras tareas. Los sistemas que controlan los automóviles, las redes sociales, los sistemas de punto de venta, el comercio electrónico, por mencionar algunos, son ejemplos de SI.

Desde la aparición del software, los SI han incorporado el software para hacer más eficiente su funcionamiento a un grado tal que se los denomina Sistemas Informáticos, acoplando la palabra “automático” a la palabra “información”.

¿Cómo se construye el Software?

El **software**, como cualquier otro producto, se construye aplicando un proceso que conduzca a un resultado de calidad, que satisfaga las necesidades de quienes lo utilizan. Un proceso de desarrollo de software es una secuencia estructurada de actividades que conduce a la obtención de un producto de software. En definitiva, un proceso define quién está haciendo qué, cuándo y cómo alcanzar un determinado objetivo. En este caso el objetivo es construir un producto de software nuevo o mejorar uno existente.



Figura 9: Proceso de Construcción del Software

Pueden identificarse **cuatro actividades fundamentales** que son comunes a todos los procesos de software:

- **Especificación del software:** donde clientes y profesionales definen el software que se construirá, sus características y las restricciones para su uso.
- **Desarrollo del software,** donde se diseña y programa el software.
- **Validación del software,** donde se controla que el software satisfaga lo que el cliente quiere.
- **Evolución del software,** donde se incorporan mejoras y nuevas características que permitirán a ese producto adaptarse a las necesidades cambiantes del cliente y el mercado.

Si consideramos las **características del software** que se explicaron anteriormente, determinamos como conclusión que el software no se obtiene por medio de un proceso de manufactura en serie o como líneas de producción, sino que para obtenerlo usamos un

proyecto, que se lo puede definir como un esfuerzo planificado, temporal y único, realizado para crear productos o servicios únicos que agreguen valor. Estos proyectos utilizan procesos que definen qué tareas deben realizar las personas que trabajan en el proyecto, para obtener los resultados deseados, utilizando como apoyo herramientas que facilitarán su trabajo. En este caso el resultado deseado es el Producto de Software.

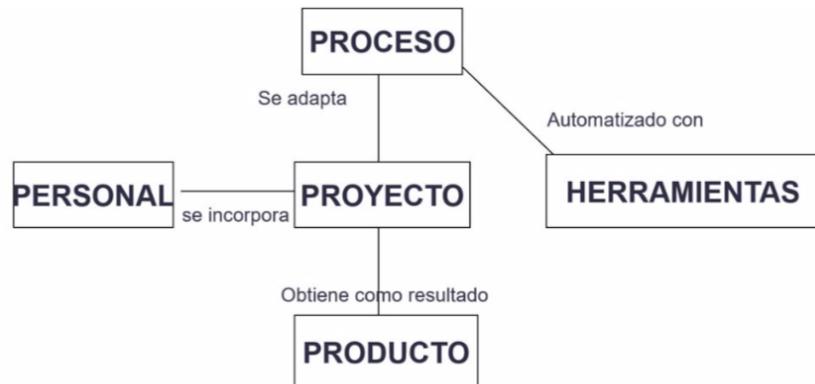


Figura 10: Relación entre Proceso, Proyecto y Producto en el desarrollo de Software

Fundamentos de programación

Diseño de Algoritmos

El hombre, en el día a día, se enfrenta constantemente a diferentes problemas que debe solucionar y para lograr solucionarlos hace uso de herramientas que le facilitan la tarea. Así, podemos pensar el uso de una calculadora para poder sumar el precio de los productos en un local y así cobrarle al cliente.

Al igual que la calculadora, la computadora también sirve para resolver problemas, pero la diferencia está en la capacidad de procesamiento de las computadoras, que hace que se puedan resolver problemas de gran complejidad, que, si los quisieramos resolver manualmente, nos llevaría mucho tiempo o ni siquiera podríamos llegar a resolverlos.

Un programador es antes que nada una persona que resuelve problemas; el programador procede a resolver un problema, a partir de la definición de un algoritmo y de la traducción de dicho algoritmo a un programa que ejecutará la computadora.

En la oración anterior se nombran algunos conceptos que debemos profundizar:

Algoritmo: un algoritmo es un método para resolver un problema, que consiste en la realización de un conjunto de pasos lógicamente ordenados tal que, partiendo de ciertos datos de entrada, permite obtener ciertos resultados que conforman la solución del problema. Así, como en la vida real, cuando tenemos que resolver un problema, o lograr un objetivo, por ejemplo: "Tengo que atarme los cordones", para alcanzar la solución de ese problema, realizamos un conjunto de pasos, de manera ordenada y secuencial. Es decir, podríamos definir un algoritmo para atarnos los cordones de la siguiente forma:

1. Ponerme las zapatillas.

-
2. Agarrar los cordones con ambas manos.
 3. Hacer el primer nudo.
 4. Hacer un bucle con cada uno de los cordones.
 5. Cruzar los dos bucles y ajustar.
 6. Corroborar que al caminar los cordones no se sueltan y la zapatilla se encuentra correctamente atada.

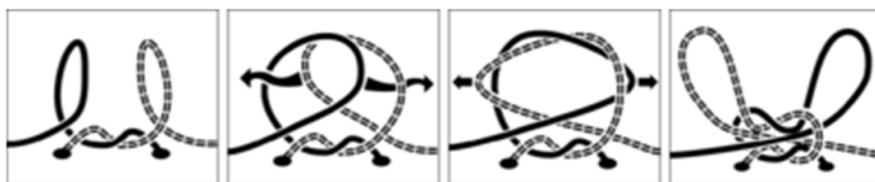


Figura 11: Algoritmo gráfico para atarse los cordones

El concepto de algoritmo es fundamental en el proceso de programación de una computadora, pero si nos detenemos a observar a nuestro alrededor, así como el ejemplo anterior podemos descubrir muchos otros: nos están dando un algoritmo cuando nos indican la forma de llegar a una dirección dada, seguimos algoritmos cuando conducimos un automóvil o cualquier tipo de vehículo. Todos los procesos de cálculo matemático que normalmente realiza una persona en sus tareas cotidianas, como sumar, restar, multiplicar o dividir, están basados en algoritmos que fueron aprendidos en la escuela primaria. Como se ve, la ejecución de algoritmos forma parte de la vida moderna.

Por otro lado, la complejidad de los distintos problemas que podamos abordar puede variar desde muy sencilla a muy compleja, dependiendo de la situación y la cantidad de elementos que intervienen. En casos de mayor complejidad suele ser una buena solución dividir al problema en diferentes subproblemas que puedan ser resueltos de manera independiente. De esta forma la solución final al problema inicial será determinada por las distintas soluciones de los problemas más pequeños cuya resolución es más sencilla.

Programa: luego de haber definido el algoritmo necesario, se debe traducir dicho algoritmo en un conjunto de instrucciones, entendibles por la computadora, que le indican a la misma lo que debe hacer; este conjunto de instrucciones conforma lo que se denomina, un programa.

Para escribir un programa se utilizan lenguajes de programación, que son lenguajes que pueden ser entendidos y procesados por la computadora. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Algoritmos

Concepto

Es un método para resolver un problema, que consiste en la realización de un conjunto de pasos lógicamente ordenados, tal que, partiendo de ciertos datos de entrada, permite obtener ciertos resultados que conforman la solución del problema.

Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar específicamente definido. Es decir, si se ejecuta un mismo algoritmo dos veces, con los mismos datos de entrada, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos. Debe tener un inicio y un final.
- Un algoritmo debe ser correcto: el resultado del algoritmo debe ser el resultado esperado.
- Un algoritmo es independiente tanto del lenguaje de programación en el que se expresa como de la computadora que lo ejecuta.

- Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
- Un algoritmo debe estar específicamente definido. Es decir, si se ejecuta un mismo algoritmo dos veces, con los mismos datos de entrada, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos. Debe tener un inicio y un final.
- Un algoritmo debe ser correcto: el resultado del algoritmo debe ser el resultado esperado.
- Un algoritmo es independiente tanto del lenguaje de programación en el que se expresa como de la computadora que lo ejecuta.

Como vimos anteriormente, el programador debe constantemente resolver problemas de manera algorítmica, lo que significa plantear el problema de forma tal que queden indicados los pasos necesarios para obtener los resultados pedidos, a partir de los datos conocidos. Lo anterior implica que un algoritmo básicamente consta de tres elementos: Datos de Entrada, Procesos y la Información de Salida.



Figura 12: Estructura de un programa, datos de entrada y sa

Cuando dicho algoritmo se transforma en un programa de computadora:

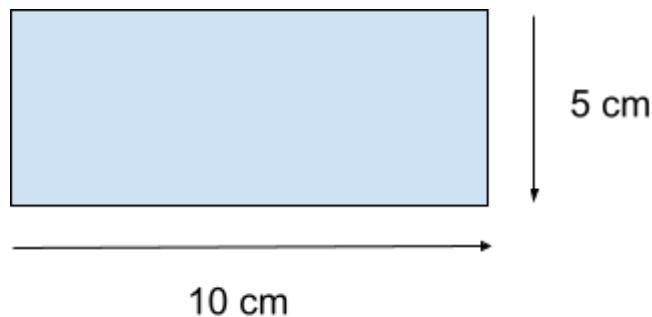
- Las **entradas** se darán por medio de un dispositivo de entrada (como los vistos en el bloque anterior), como pueden ser el teclado, disco duro, teléfono, etc. Este proceso se lo conoce como entrada de datos, operación de lectura o acción de leer.
- Las **salidas de datos** se presentan en dispositivos periféricos de salida, que pueden ser pantalla, impresora, discos, etc. Este proceso se lo conoce como salida de datos, operación de escritura o acción de escribir.

Dado un problema, para plantear un algoritmo que permita resolverlo, es conveniente entender correctamente la situación problemática y su contexto, tratando de deducir del mismo los elementos ya indicados (entradas, procesos y salida). En este sentido entonces, para crear un algoritmo:

1. Comenzar identificando los resultados esperados, porque así quedan claros los objetivos a cumplir.
2. Luego, individualizar los datos con que se cuenta y determinar si con estos datos es suficiente para llegar a los resultados esperados. Es decir, definir los datos de entrada con los que se va a trabajar para lograr el resultado.
3. Finalmente si los datos son completos y los objetivos claros, se intentan plantear los procesos necesarios para pasar de los datos de entrada a los datos de salida.

Para comprender esto, veamos un ejemplo:

Problema:



Obtención del área de un rectángulo:

1. Resultado esperado: área del rectángulo.

Salida: área

Fórmula del área: base x altura.

-
2. Los datos con los que se dispone, es decir las entradas de datos son:

Dato de **Entrada 1**: altura: 5 cm

Dato de **Entrada 2**: base: 10 cm

3. El **proceso** para obtener el área del rectángulo es:

área=base*altura

área=50

Herramientas para la representación gráfica de los algoritmos

Como se especificó anteriormente, un algoritmo es independiente del lenguaje de programación que se utilice. Es por esto, que existen distintas **técnicas de representación** de un algoritmo que permiten esta diferenciación con el lenguaje de programación elegido. De esta forma el algoritmo puede ser representado en **cualquier lenguaje**. Existen diversas herramientas para representar gráficamente un algoritmo. En este material presentaremos dos:

1. Diagrama de flujo.
2. Lenguaje de especificación de algoritmos: pseudocódigo.

Diagramas de Flujo

Un diagrama de flujo hace uso de símbolos estándar que unidos por flechas, indican la secuencia en que se deben ejecutar.

Algunos de estos símbolos son:

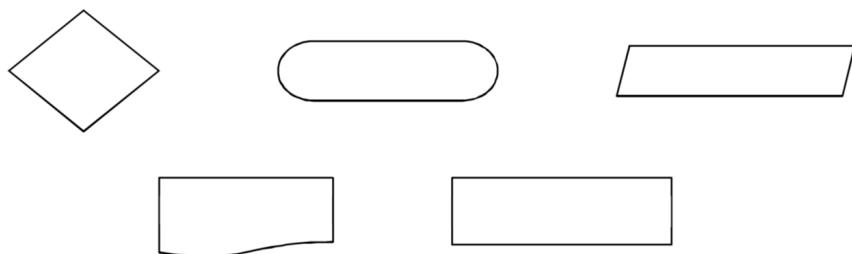


Figura 13: Símbolos utilizados en Diagramas de Flujo

Pseudocódigo

Conocido como lenguaje de especificación de algoritmos, el pseudocódigo tiene una estructura muy similar al lenguaje natural y sirve para poder expresar algoritmos y programas de forma independiente del lenguaje de programación. Además, es muy utilizado para comunicar y representar ideas que puedan ser entendidas por programadores que conozcan distintos lenguajes. El pseudocódigo luego se traduce a un lenguaje de programación específico ya que la computadora no puede ejecutar el pseudocódigo. Su uso tiene ventajas porque permite al programador una mejor concentración de la lógica y estructuras de control y no preocuparse de las reglas de un lenguaje de programación específico.

Un **ejemplo básico de pseudocódigo**, considerando el ejemplo utilizado anteriormente, es el siguiente:

```
INICIO FUNCION CALCULAR_AREA
    DEFINIR BASE: 5
    DEFINIR ALTURA: 10
    DEFINIR AREA: BASE * ALTURA
    DEVOLVER AREA
FIN
```

Lenguajes de Programación

Concepto

Los lenguajes de programación son lenguajes que pueden ser entendidos y procesados por la computadora. En otras palabras podemos decir que los lenguajes de programación son todos los símbolos, caracteres y reglas de uso que permiten a las personas "comunicarse" con las computadoras.

Un lenguaje de programación es un **sistema estructurado y diseñado** principalmente para que las computadoras se entiendan entre sí y con nosotros, los humanos. Contiene un conjunto de acciones consecutivas que el ordenador 2 debe ejecutar.

Estos lenguajes de programación usan diferentes normas o bases y se utilizan para controlar cómo se comporta una máquina (por ejemplo, un ordenador), también pueden usarse para crear programas informáticos que formarán productos de software.

El término "**programación**" se define como un proceso por medio del cual se diseña, se codifica, se escribe, se prueba y se depura un código básico para las computadoras. Ese código es el que se llama "código fuente" que caracteriza a cada lenguaje de programación. Cada lenguaje de programación tiene un "código fuente" característico y único que está diseñado para una función o un propósito determinado y que nos sirven para que una computadora se comporte de una manera deseada.

En la actualidad existe un gran número de lenguajes de programación diferentes³ . Algunos, denominados **lenguajes específicos de dominio** (o con las siglas en inglés, DSL, Domain Specific Languages) se crean para una aplicación especial, mientras que otros son herramientas de uso general, más flexibles, que son apropiadas para muchos tipos de aplicaciones. En todo caso los lenguajes de programación deben tener instrucciones que pertenecen a las categorías ya familiares de entrada/salida, cálculo/manipulación de textos, lógica/comparación y almacenamiento / recuperación.

A continuación, presentamos un recorrido en el tiempo por los lenguajes de programación más conocidos:

³ https://es.wikipedia.org/wiki/Anexo:3ALenguajes_de_programaci%C3%B3n, este sitio lista en orden alfabético los lenguajes de programación existentes, tanto de uso actual como histórico.

1957 - 1959

- Fortran (Formula Translation)
- LISP (List Processor)
- COBOL (Common Business-Oriented Language)

Considerados los lenguajes **más viejos** utilizados hoy en día. Son lenguajes de alto nivel que fueron creados por científicos, matemáticos y empresarios de la computación.

Principales usos: Aplicaciones científicas y de ingeniería para supercomputadoras, desarrollo de Inteligencia Artificial, software empresarial.

1970

Pascal (nombrado así en honor al matemático y físico Francés Blaise Pascal)



Lenguaje de alto nivel, utilizado para la enseñanza de la programación estructurada y la estructuración de datos. Las versiones comerciales de Pascal fueron ampliamente utilizadas en los años 80's.

Creador: Niklaus Wirth

Principales usos: Enseñanza de la programación. Objeto Pascal, un derivado, se utiliza comúnmente para el desarrollo de aplicaciones Windows.

Usado por: Apple Lisa (1983) y Skype.

1972

C (Basado en un programa anterior llamado “B”)



Lenguaje de propósito general, de bajo nivel. Creado por Unix Systems, en Bell Labs. Es el lenguaje más popular (precedido por Java). De él se derivan muchos lenguajes como C#, Java, Javascript, Perl, PHP y Python.

Creador: Dennis Ritchie (Laboratorios Bell)

Principales usos: Programación multiplataforma, programación de sistemas, programación en Unix y desarrollo de videojuegos. Usado por: Unix (reescrito en C en 1973), primeros servidores y clientes de la WWW.

1983

C++ (Originariamente “C con clases”; ++ es el operador de incremento en “C”)

	<p>Lenguaje multiparadigma. Una extensión de C con mejoras como clases, funciones virtuales y plantillas.</p> <p>Creador: Bjarne Stroustrup (Laboratorios Bell)</p> <p>Principales usos: Desarrollo de aplicaciones comerciales, software embebido, aplicaciones cliente-servidor en videojuegos.</p> <p>Usado por: Adobe, Google Chrome, Mozilla Firefox, Microsoft Internet Explorer.</p>
---	---

Objective-C (Object-oriented extension de “C”)

 Objective-C	<p>Lenguaje de propósito general, de alto nivel. Ampliado en C, adicionaba una funcionalidad de paso de mensajes. Se hizo muy popular por ser el lenguaje preferido para el desarrollo de aplicaciones para productos de Apple en los últimos años hasta ser reemplazado por Swift.</p> <p>Creador: Brad Cox y Tom Love (Stepstone)</p>
---	---

	<p>Principales usos: Programación Apple. Usado por: Apple OS X y sistemas operativos iOS</p>
--	--

1987

Perl ("Pearl" ya estaba ocupado)

	<p>Lenguaje de propósito general, de alto nivel, muy poderoso en el manejo de expresiones regulares. Creado para el procesamiento de reportes en sistemas Unix. Hoy en día es conocido por su alto poder y versatilidad.</p> <p>Creador: Larry Wall (Unisys)</p> <p>Principales usos: Imágenes generadas por computadora, aplicaciones de base de datos, administración de sistemas, programación web y programación de gráficos.</p> <p>Usado por: IMDb, Amazon, Priceline, Ticketmaster</p>
--	--

1991

Python (en honor a la compañía de comedia británica Monty Python)



Lenguaje de propósito general, de alto nivel. Creado para apoyar una gran variedad de estilos de programación de manera divertida. Muchos tutoriales, ejemplos de código e instrucciones a menudo contienen referencias a Monty Python.

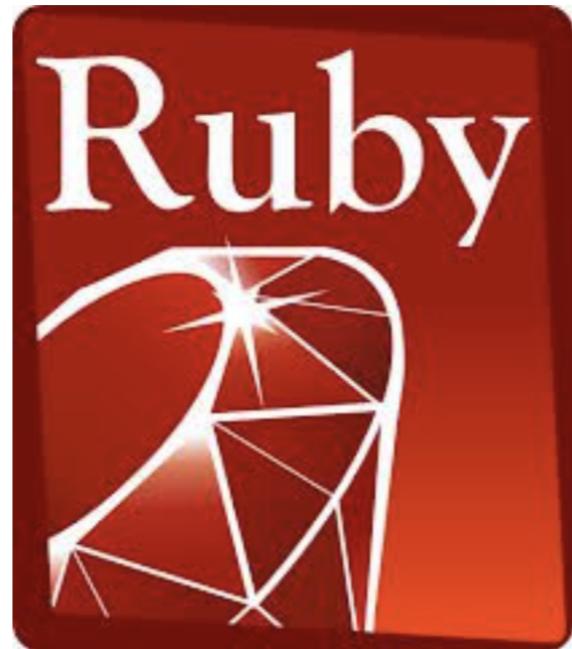
Creador: Guido Van Rossum (CWI)

Principales usos: Aplicaciones Web, desarrollo de software, seguridad informática.

Usado por: Google, Yahoo, Spotify

1993

Ruby (La piedra del zodiaco de uno de los creadores).



Lenguaje de propósito general, de alto nivel. Un programa de enseñanza, influenciado por Perl, Ada, Lisp, Smalltalk, entre otros. Diseñado para hacer la programación más productiva y agradable.

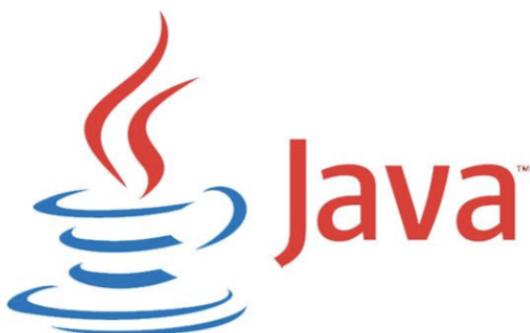
Creador: Yukihiro Matsumoto

Principales usos: Desarrollo de aplicaciones Web, Ruby on Rails.

Usado por: Twitter, Hulu, Groupon.

1995

Java (inspirado en las tazas de café consumidas mientras se desarrollaba el lenguaje).



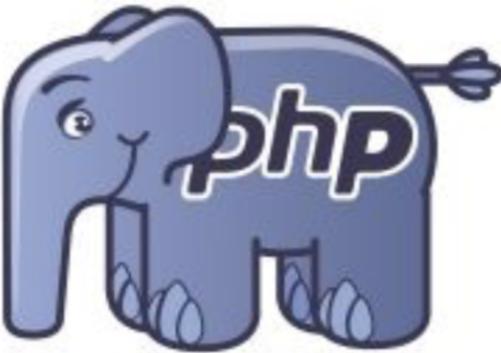
Lenguaje de propósito general, de alto nivel. Hecho para un proyecto de televisión interactiva. Funcionalidad de programación multiplataforma. Es actualmente el lenguaje de programación más popular en el mundo⁴.

Creador: James Gosling (Sun Microsystems)

Principales usos: Programación Web,

	desarrollo de aplicaciones Web, desarrollo de software, desarrollo de interfaz gráfica de usuario. Usado por: Android OS/Apps
--	--

PHP (Formalmente: "Personal Home Page", ahora es por "Hypertext Preprocessor").

	Lenguaje de código abierto, de propósito general. Se utiliza para construir páginas web dinámicas. Más ampliamente usado en software de código abierto para empresas. Creador: Rasmus Lerdorf Principales usos: Construcción y mantenimiento de páginas web dinámicas, desarrollo del lado del servidor. Usado por: Facebook, Wikipedia, Digg, WordPress, Joomla.
--	---

Javascript

 JavaScript	<p>Lenguaje de alto nivel. Creado para extender las funcionalidades de las páginas web. Usado por páginas dinámicas para el envío y validación de formularios, interactividad, animación, seguimiento de actividades de usuario, etc.</p> <p>Creador: Brendan Eich (Netscape)</p> <p>Principales usos: Desarrollo de web dinámica, documentos PDF, navegadores web y widgets de Escritorio. Usado por: Gmail, Adobe Photoshop, Mozilla Firefox.</p>
--	---

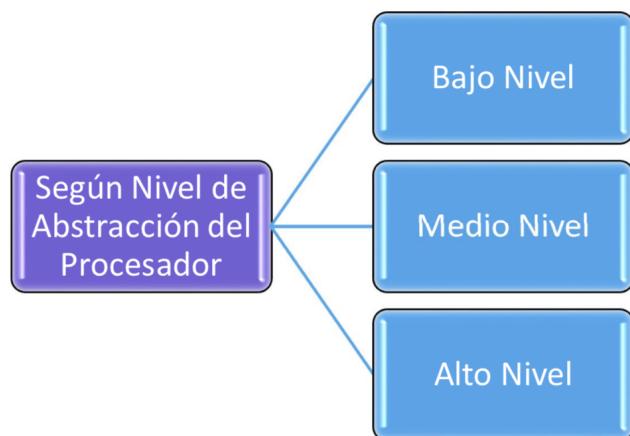
Tipos de Lenguajes de Programación

Para conocer un poco más sobre los lenguajes de programación analizaremos algunas clasificaciones posibles:



Nivel de Abstracción del Procesador

Según el nivel de abstracción del procesador, los lenguajes de programación, se clasifican en:



Paradigma de Programación

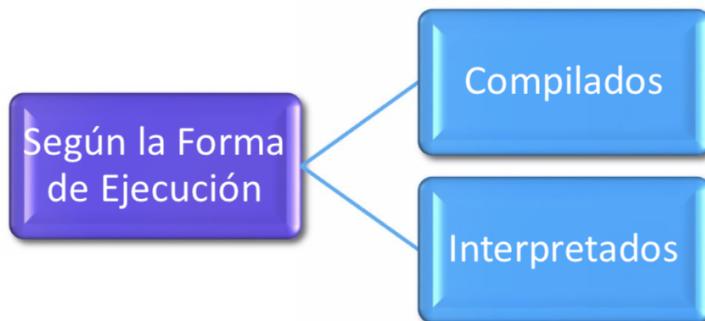
Según el Paradigma de Programación, los lenguajes se clasifican en:



La descripción de cada uno de los paradigmas tiene una sección específica, más adelante en este material.

Forma de Ejecución

Según la Forma de Ejecución, los lenguajes de programación, se clasifican en:



Lenguajes Compilados

Los compiladores son aquellos programas cuya función es traducir un programa escrito en un determinado lenguaje a un idioma que la computadora entienda (lenguaje máquina con código binario). Al usar un lenguaje compilado, el programa desarrollado es controlado previamente, por el compilador, y por eso nunca se ejecuta si tiene errores de código. Es decir, se compila y si la compilación es exitosa ese programa se puede ejecutar.

Un programa compilado es aquel cuyo código fuente, escrito en un lenguaje de alto nivel, es traducido por un compilador a un archivo ejecutable entendible para la máquina en determinada plataforma. Con ese archivo se puede ejecutar el programa cuantas veces sea necesario sin tener que repetir el proceso por lo que el tiempo de espera entre ejecución y ejecución es ínfimo.

Dentro de los lenguajes de programación que son **compilados** tenemos la familia C que incluye a C++, Objective C, C# y también otros como Fortran, Pascal, Haskell y Visual Basic.

Java es un caso particular ya que hace uso de una máquina virtual que se encarga de la traducción del código fuente por lo que hay veces que es denominado compilado e interpretado. Otra ventaja de la máquina virtual que usa Java, es que le permite ejecutar código Java en cualquier máquina que tenga instalada la JVM (Java Virtual Machine).

Lenguajes Interpretados

Básicamente un lenguaje interpretado es aquel en el cual sus instrucciones o más bien el código fuente, escrito por el programador en un lenguaje de alto nivel, es traducido por el intérprete a un lenguaje entendible para la máquina paso a paso, instrucción por instrucción. El proceso se repite cada vez que se ejecuta el programa el código en cuestión.

Estos lenguajes utilizan una alternativa diferente de los compiladores para traducir lenguajes de alto nivel. En vez de traducir el programa fuente y grabar en forma permanente el código objeto que se produce durante la corrida de compilación para utilizarlo en una corrida de producción futura, el programador sólo carga el programa fuente en la computadora junto con los datos que se van a procesar. A continuación, un programa intérprete, almacenado en el sistema operativo del disco, o incluido de manera permanente dentro de la máquina, convierte cada proposición del programa fuente en lenguaje de máquina conforme vaya siendo necesario durante el proceso de los datos. No se graba el código objeto para utilizarlo posteriormente.

El uso de los lenguajes interpretados ha venido en crecimiento y cuyos máximos representantes son los lenguajes usados para el desarrollo web entre estos Ruby, Python, PHP, **JavaScript** y otros como Perl, Smalltalk, MATLAB, Mathematica.

Los lenguajes interpretados permiten el tipado dinámico de datos, es decir, no es necesario inicializar una variable con determinado tipo de dato, sino que esta puede cambiar su tipo en condición al dato que almacene entre otras características más.

También tienen por ventaja una gran independencia de la plataforma donde se ejecutan de ahí que los tres primeros mencionados arriba sean multiplataforma comparándolos con algunos lenguajes compilados como C#, y los programas escritos en lenguajes interpretados utilizan menos recursos de hardware (más livianos).

La principal desventaja de estos lenguajes es el tiempo que necesitan para ser interpretados. Al tener que ser traducido a lenguaje máquina con cada ejecución, este proceso es más lento que en los lenguajes compilados; sin embargo, algunos lenguajes

poseen una máquina virtual que hace una traducción a lenguaje intermedio con lo cual el traducirlo a lenguaje de bajo nivel toma menos tiempo.

En este curso abordaremos como lenguaje de programación Javascript ya que ha tenido un crecimiento exponencial durante los últimos años y se proyecta que es uno de los lenguajes que más proyección y salida laboral va a tener en los próximos años. Otro motivo es que si nos dedicamos a hacer desarrollo web o móvil inclusive, aprender Javascript representa prácticamente una obligación.

Tipos de Lenguajes de Programación

Los principales tipos de lenguajes utilizados en la actualidad son tres:

- **Lenguajes máquina**
- **Lenguaje de bajo nivel (ensamblador)**
- **Lenguajes de alto nivel**

La elección del lenguaje de programación a utilizar depende mucho del objetivo del software. Por ejemplo, para desarrollar aplicaciones que deben responder en tiempo real como por ejemplo, el control de la velocidad crucero en un sistema de navegación de un auto, debemos tener mayor control del hardware disponible, por lo que privilegiaremos lenguajes de más bajo nivel que nos permitan hacer un uso más eficiente de los recursos. En cambio, para aplicaciones de web como sistemas de gestión de productos, calendarios, correo electrónico, entre otras privilegiaremos la elección de lenguajes de más alto nivel que nos permitan ser más eficientes en cuanto a la codificación ya que, en términos generales, es necesario escribir menos líneas de código en los lenguajes de alto nivel, que para su equivalente en bajo nivel.

El lenguaje máquina

Los lenguajes máquina son aquellos que están escritos en lenguajes cuyas instrucciones son cadenas binarias (cadenas o series de caracteres -dígitos- 0 y 1) que especifican una operación, y las posiciones (dirección) de memoria implicadas en la operación se denominan instrucciones de máquina o código máquina. El código máquina es el conocido código binario.

En los primeros tiempos del desarrollo de los ordenadores era necesario programarlos directamente de esta forma, sin embargo, eran máquinas extraordinariamente limitadas, con muy pocas instrucciones por lo que aún era posible; en la actualidad esto es completamente irrealizable por lo que es necesario utilizar lenguajes más fácilmente comprensibles para los humanos que deben ser traducidos a código máquina para su ejecución.

Ejemplo de una instrucción:

1110 0010 0010 0001 0000 0000 0010 0000

El lenguaje de bajo nivel

Los lenguajes de bajo nivel son **más fáciles** de utilizar que los lenguajes máquina, pero, al igual, que ellos, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el ensamblador o assembler. Las instrucciones en lenguaje ensamblador son instrucciones conocidas como mnemotécnicas (mnemonics). Por ejemplo, nemotécnicos típicos de operaciones aritméticas son: en inglés, ADD, SUB, DIV, etc.; en español, SUM, para sumar, RES, para restar, DIV, para dividir etc.

Lenguajes de alto nivel

Los lenguajes de alto nivel son **los más utilizados por los programadores**. Están diseñados para que las personas escriban y entiendan los programas de un modo mucho más fácil que los lenguajes máquina y ensambladores. Otra razón es que un programa escrito en lenguaje de alto nivel es independiente de la máquina; esto es, las instrucciones del programa de la computadora no dependen del diseño del hardware o de una computadora en particular. En consecuencia, los programas escritos en lenguaje de alto nivel son portables o transportables, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras; al contrario que los programas en lenguaje máquina o ensamblador, que sólo se pueden ejecutar en un determinado tipo de computadora. Esto es posible porque los lenguajes de alto nivel son traducidos a lenguaje máquina por un tipo de programa especial denominado "compilador". Un compilador toma como entrada un algoritmo escrito en un lenguaje de alto nivel y lo convierte a instrucciones inteligibles por el ordenador; los compiladores deben estar adaptados a cada tipo de ordenador pues deben generar código máquina específico para el mismo.

Ejemplos de Lenguajes de Alto Nivel:

C, C++, Java, Python, Go, Swift, C#, PHP, JavaScript

¿Qué es un Programa?

Es un algoritmo escrito en algún lenguaje de programación de computadoras.

Pasos para la construcción de un programa**1. DEFINICIÓN DEL PROBLEMA**

En este paso se determina la información inicial para la elaboración del programa. Es donde se determina qué es lo que debe resolverse con el computador, el cual requiere una definición clara y precisa. Es importante que se conozca lo que se desea que realice la computadora; mientras la definición del problema no se conozca del todo, no tiene mucho caso continuar con la siguiente etapa.

2. ANÁLISIS DEL PROBLEMA

Una vez que se ha comprendido lo que se desea de la computadora, es necesario definir:

- Los datos de entrada.
- Los datos de salida
- Los métodos y fórmulas que se necesitan para procesar los datos.

Una recomendación muy práctica es la de colocarse en el lugar de la computadora y analizar qué es lo que se necesita que se ordene y en qué secuencia para producir los resultados esperados.

3.DISEÑO DEL ALGORITMO

Se puede utilizar algunas de las herramientas de representación de algoritmos mencionadas anteriormente. Este proceso consiste en definir la secuencia de pasos que se deben llevar a cabo para conseguir la salida identificada en el paso anterior.

4.CODIFICACIÓN

La codificación es la operación de escribir la solución del problema (de acuerdo a la lógica del diagrama de flujo o pseudocódigo), en una serie de instrucciones detalladas, en un código reconocible por la computadora. La serie de instrucciones detalladas se conoce como código fuente, el cual se escribe en un lenguaje de programación o lenguaje de alto nivel.

5. PRUEBA Y DEPURACIÓN

Se denomina prueba de escritorio a la comprobación que se hace de un algoritmo para saber si está bien realizado. Esta prueba consiste en tomar datos específicos como entrada y seguir la secuencia indicada en el algoritmo hasta obtener un resultado, el análisis de estos resultados indicará si el algoritmo está correcto o si por el contrario hay necesidad de corregirlo o hacerle ajustes.

Elementos de un Programa

Variables y Constantes

A la hora de elaborar un programa es necesario usar datos; en el caso del ejemplo del cálculo del área del rectángulo, para poder obtener el área del mismo, necesitamos almacenar en la memoria de la computadora el valor de la base y de la altura, para luego poder multiplicar sus valores.

Recordemos que no es lo mismo grabar los datos en memoria que grabarlos en el disco duro. Cuando decimos grabar en memoria nos estaremos refiriendo a grabar esos datos en la RAM. Ahora bien, para grabar esos datos en la RAM podemos hacerlo utilizando dos elementos, llamados: variables y constantes. Los dos elementos funcionan como fuentes de almacenamiento de datos, la gran diferencia entre los dos es que en el caso de las constantes su valor dado no varía en el transcurso de todo el programa.

Podría decirse que tanto las variables como las constantes, son direcciones de memoria con un valor, ya sea un número, una letra, o valor nulo (cuando no tiene valor alguno, se denomina valor nulo). Estos elementos permiten almacenar temporalmente datos en la

computadora para luego poder realizar cálculos y operaciones con los mismos. Al almacenarlos en memoria, podemos nombrarlos en cualquier parte de nuestro programa y obtener el valor del dato almacenado, se dice que la variable nos devuelve el valor almacenado.

A continuación, se muestra un esquema, que representa la memoria RAM, como un conjunto de filas, donde, en este caso, cada fila representa un byte y tiene una dirección asociada.

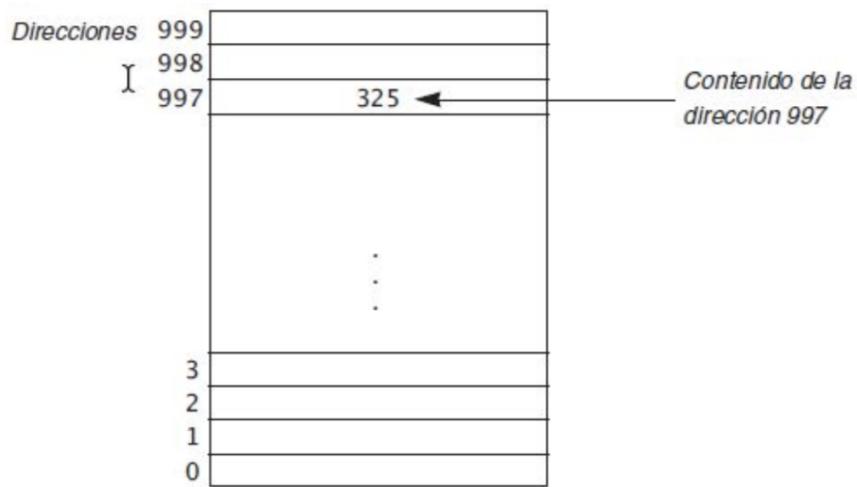


Figura 14: Representación de la memoria RAM

Variables

Son elementos de almacenamiento de datos. Representan una dirección de memoria en donde se almacena un dato, que puede variar en el desarrollo del programa. Una variable es un grupo de bytes asociado a un nombre o identificador, y a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a esa variable.

En una variable se puede almacenar distintos tipos de datos. De acuerdo al tipo de dato, definido para cada lenguaje de programación, será la cantidad de bytes que ocupa dicha variable en la memoria.

Type	Size	Range
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	$-3.40282347 \times 10^{38}$ to 3.4028347×10^{38}
double	8 bytes	$-1.79769313486231570 \times 10^{308}$ to $1.79769313486231570 \times 10^{308}$
char	2 bytes	one character
String	2 or more bytes	one or more characters

Tabla 1: Tabla de equivalencias de las unidades de memoria

Lo más importante de la definición de las variables y la elección del tipo de datos asociados es el significado de la variable, o su semántica: ya que en base al tipo de datos seleccionado serán las operaciones que podamos realizar con esa variable, por ejemplo: si tenemos la variable edad deberíamos seleccionar un tipo de datos como integer (número entero) ya que las operaciones relacionadas serán de comparación, sumas o restas y no es necesario tener una profundidad de decimales.

A continuación, se listan algunos de los tipos de datos más comunes y sus posibles usos que existen en los lenguajes de programación:

Tipo de Datos	Significado	Ejemplos de uso
Byte	Número entero de 8 bits. Con signo	Temperatura de una habitación en grados Celsius
Short	Número entero de 16 bits. Con signo	Edad de una Persona
Int	Número entero de 32 bits. Con signo	Distancia entre localidades medida en metros

Long	Número entero de 64 bits. Con signo	Producto entre dos distancias almacenadas en variables tipo int como la anterior
Float	Número Real de 32 bits.	Altura de algún objeto
Double	Número Real de 64 bits.	Proporción entre dos magnitudes
Boolean	Valor lógico:true (verdadero) o false (falso)	Almacenar si el usuario ha aprobado un examen o no

Para algunos lenguajes de programación es posible especificar si las variables numéricas tendrán o no signo (es decir que puedan tomar valores negativos y positivos, o sólo positivos). La elección de tener o no signo (definidas con la palabra reservada `unsigned`) depende del significado de la variable ya que al no tenerlo podremos almacenar el doble de valores, por ejemplo, una variable `short` sin signo posee valores desde el 0 al 65535.

Constantes

Elementos de almacenamiento de datos. Representan una dirección de memoria en donde se almacena un dato pero que no varía durante la ejecución del programa. Se podría pensar en un ejemplo de necesitar utilizar en el programa el número pi, como el mismo no varía, se puede definir una constante pi y asignarle el valor 3.14.

Operadores

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Los operadores son símbolos especiales que sirven para ejecutar una determinada operación, devolviendo el resultado de la misma.

Para comprender lo que es un operador, debemos primero introducir el concepto de Expresión. Una expresión es, normalmente, una ecuación matemática, tal como $3 + 5$. En esta expresión, el símbolo más (+) es el operador de suma, y los números 3 y 5 se llaman operandos. En síntesis, una expresión es una secuencia de operaciones y operandos que especifica un cálculo.

Existen diferentes tipos de operadores:

1. Operador de asignación

Es el operador más simple que existe, se utiliza para asignar un valor a una variable o a una constante. El signo que representa la asignación es el = y este operador indica que el valor a la derecha del = será asignado a lo que está a la izquierda del mismo.

Ejemplo en pseudocódigo:

Entero edad = 20

Decimal precio = 25.45

2. Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (+), resta (-), multiplicación (*), división (/) y resto de la división entera (%), por ejemplo $50\%8=2$ porque necesitamos obtener un número entero que queda luego de determinar la cantidad de veces que 8 entra en 50.

Ejemplo:

Expresión	Operador	Operandos	Resultado arrojado
$5 * 7$	*	5 , 7	35
$6 + 3$	+	6 , 3	9
$20 - 4$	-	20 , 4	16
$50 \% 8$	%	50 , 8	2
$45/5$	/	45,5	9

Tabla Resumen Operadores Aritméticos:

Operador	Significado
+	Suma
-	Resta
*	Producto
/	División
%	Resto de la división entera

3. Los operadores unitarios

Los operadores unitarios requieren sólo un operando; que llevan a cabo diversas operaciones, tales como incrementar/decrementar un valor de a uno, negar una expresión, o invertir el valor de un booleano.

Operador	Descripción	Ejemplo	Resultado
<code>++</code>	operador de incremento; incrementa un valor de a 1	<code>int suma=20; suma++;</code>	<code>suma=21</code>
<code>--</code>	operador de decremento; Reduce un valor de a 1	<code>int resta=20; resta--;</code>	<code>resta=19</code>
<code>!</code>	operador de complemento lógico; invierte el valor de un valor booleano	<code>boolean a=true; boolean b= !a;</code>	<code>b=false</code>

4. Operadores Condicionales

Son aquellos operadores que sirven para comparar valores. Siempre devuelven valores booleanos: TRUE O FALSE. Pueden ser Relacionales o Lógicos.

5. Operadores relacionales

Los operadores relacionales sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor.

Los operadores relacionales determinan si un operando es mayor que, menor que, igual a, o no igual a otro operando. La mayoría de estos operadores probablemente le resultará familiar. Tenga en cuenta que debe utilizar " == ", no " = ", al probar si dos valores primitivos son iguales.

Operador	Significado
<code>==</code>	Igual a
<code>!=</code>	No igual a
<code>></code>	Mayor que
<code>>=</code>	Mayor o igual que
<code><</code>	Menor que
<code><=</code>	Menor o igual que

Expresión	Operador	Resultado
<code>a > b</code>	<code>></code>	true: si a es mayor que b false: si a es menor que b
<code>a >= b</code>	<code>>=</code>	true: si a es mayor o igual que b false: si a es menor que b
<code>a < b</code>	<code><</code>	true: si a es menor que b false: si a es mayor que b
<code>a <= b</code>	<code><=</code>	true: si a es menor o igual que b false: si a es mayor que b.
<code>a == b</code>	<code>==</code>	true: si a y b son iguales. false: si a y b son diferentes.
<code>a != b</code>	<code>!=</code>	true: si a y b son diferentes false: si a y b son iguales.

6. Operadores lógicos

Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. Se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales.

Expresión	Nombre Operador	Operador	Resultado
a && b	AND	&&	true: si a y b son verdaderos. false: si a es falso, o si b es falso, o si a y b son falsos
a b	OR		true: si a es verdadero, o si b es verdadero, o si a y b son verdaderos. false: si a y b son falsos.

Debe notarse que en ciertos casos el segundo operando no se evalúa porque no es necesario (si ambos tienen que ser true y el primero es false ya se sabe que la condición de que ambos sean true no se va a cumplir).

Rutinas

Las rutinas son uno de los recursos más valiosos cuando se trabaja en programación ya que permiten que los programas sean más simples, debido a que el programa principal se compone de diferentes rutinas donde cada una de ellas realiza una tarea determinada.

Una rutina se define como un bloque, formado por un conjunto de instrucciones que realizan una tarea específica y a la cual se la puede llamar desde cualquier parte del programa principal. Además, una rutina puede opcionalmente tener un **valor de retorno y parámetros**. El valor de retorno puede entenderse como el resultado de las instrucciones llevadas a cabo por la rutina, por ejemplo si para una rutina llamada `sumar(a, b)` podríamos esperar que su valor de retorno sea la suma de los números a y b. En el caso anterior, a y b son los datos de entrada de la rutina necesarios para realizar los cálculos correspondientes. A estos datos de entrada los denominamos **parámetros** y a las rutinas que reciben parámetros las

denominamos **funciones**, **procedimientos** o **métodos**, dependiendo del lenguaje de programación.

Ejemplos de rutinas:

1. *sumarPrecioProductos(precioProducto1, precioProducto2)*

Rutina que realiza la suma de los precios de los productos comprados por un cliente y devuelve el monto total conseguido.

2. *aplicarDescuento(montoTotal)*

Rutina que a partir de un monto total aplica un descuento de 10% y devuelve el monto total con el descuento aplicado.

3. *añadirProductoACarrito(codigoProducto)*

Rutina que permite agregar un producto a un carrito de compras, la misma recibe como parámetro el código de un producto.

Entonces nuestro programa puede hacer uso de dichas rutinas cuando lo necesite. Por ejemplo cuando un cliente realice una compra determinada, podemos llamar a la rutina *sumarPrecioProductos* y cobrarle el monto devuelto por la misma. En el caso que el cliente abonara con un cupón de descuento, podemos entonces llamar a la rutina *aplicarDescuento* y así obtener el nuevo monto con el 10% aplicado.

Estructuras de control

Un programa puede ser escrito utilizando tres tipos de estructuras de control:

- a) secuenciales
- b) selectivas o de decisión
- c) repetitivas

Las Estructuras de Control determinan el orden en que deben ejecutarse las instrucciones de un algoritmo: si serán recorridas una luego de la otra, si habrá que tomar decisiones sobre si ejecutar o no alguna acción o si habrá repeticiones.

Estructura secuencial

Es la estructura en donde una acción (instrucción) sigue a otra de manera secuencial. Las tareas se dan de tal forma que la salida de una es la entrada de la que sigue y así en lo sucesivo hasta cumplir con todo el proceso. Esta estructura de control es la más simple, permite que las instrucciones que la constituyen se ejecuten una tras otra en el orden en que se listan. Por ejemplo, considérese el siguiente fragmento de un algoritmo:

En este fragmento se indica que se ejecute la operación 1 y a continuación la operación



Figura 15: Diagrama de Flujo Secuencial

Estructura alternativa

Estas estructuras de control son de gran utilidad para cuando el algoritmo a desarrollar requiera una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existe un número de posibles alternativas que resultan de la evaluación de una determinada condición.

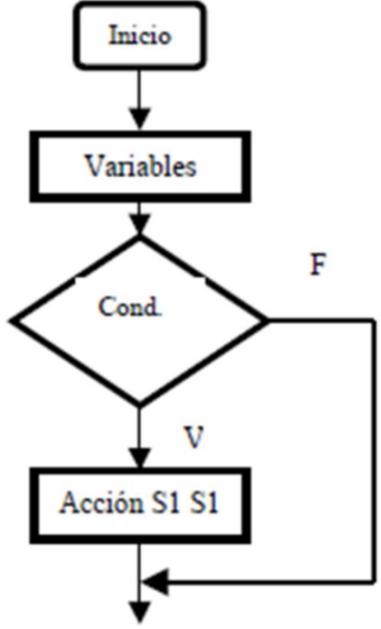
Este tipo de estructuras son utilizadas para tomar decisiones lógicas, es por esto que también se denominan estructuras de decisión o selectivas.

En estas estructuras, se realiza una evaluación de una condición y de acuerdo al resultado, el algoritmo realiza una determinada acción. Las condiciones son especificadas utilizando expresiones lógicas.

Las estructuras selectivas/alternativas pueden ser:

- Simples
- Dobles
- Múltiples

Alternativa simple (si-entonces/if-then)

 <p>Figura 16: Diagrama de Flujo de alternativa simple</p>	<p>La estructura alternativa simple si-entonces (en inglés if-then) lleva a cabo una acción al cumplirse una determinada condición. La selección si-entonces evalúa la condición y:</p> <ul style="list-style-type: none"> • Si la condición es verdadera, ejecuta la acción S1 • Si la condición es falsa, no ejecuta nada.
<p><u>En español:</u> Si <condición> entonces <acción S1> Fin_si</p>	<p><u>En Inglés:</u> If <condition> then <action S1> end_if</p>

Ejemplo:
INICIO

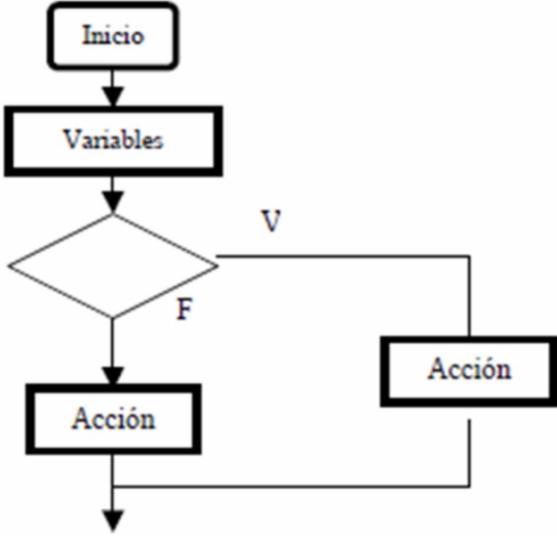
ENTERO edad = 18

SI (edad > 18) ENTONCES:

puede manejar un auto

FIN_SI

FIN
Alternativa Doble (si-entonces-sino/if-then-else)

 <p>Figura 17: Diagrama de Flujo de alternativa doble</p>	<p>Existen limitaciones en la estructura anterior, y se necesitará normalmente una estructura que permita elegir dos opciones o alternativas posibles, de acuerdo al cumplimiento o no de una determinada condición:</p> <ul style="list-style-type: none"> • Si la condición es verdadera, se ejecuta la acción S1 • Si la condición es falsa, se ejecuta la acción S2
<p>En español:</p> <pre>Si <condición> entonces <acción S1> sino <acción S2> Fin_Si</pre>	<p>En inglés:</p> <pre>If <condición> then<acción> else<acción S2> End_if</pre>

Ejemplo:
INICIO

BOOLEANO afueraLlueve = verdadero

SI (afueraLlueve es verdadero)

ENTONCES:

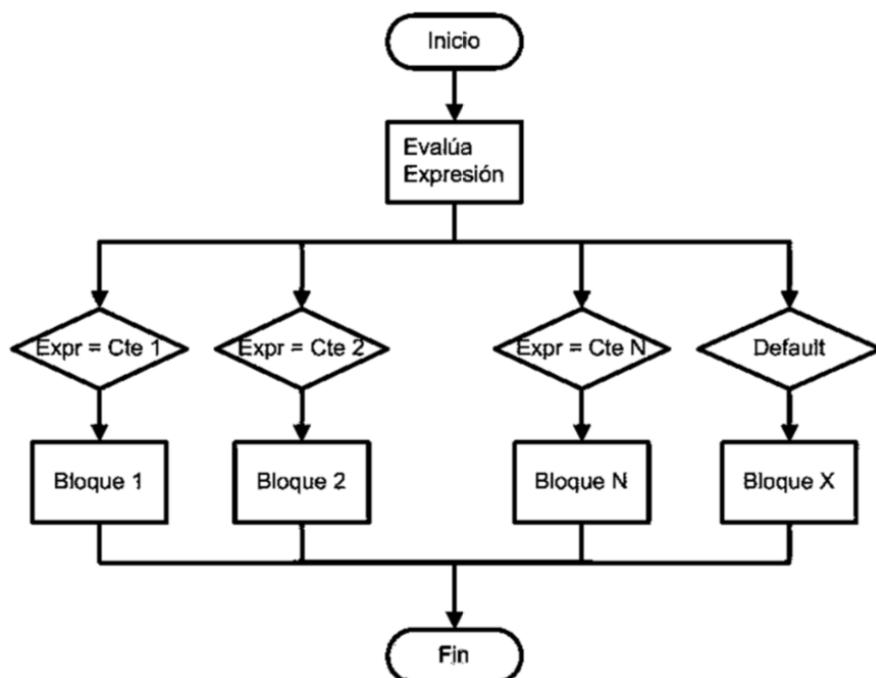
me quedo viendo películas

SINO:

salgo al parque a tomar mates

FIN_SI
FIN
Alternativa de Decisión múltiple (según_sea, caso de/case)

Se utiliza cuando existen más de dos alternativas para elegir. Esto podría solucionarse por medio de estructuras alternativas simples o dobles, anidadas o en cascada. Sin embargo, se



pueden plantear serios problemas de escritura del algoritmo, de comprensión y de legibilidad, si el número de alternativas es grande.

En esta estructura, se evalúa una condición o expresión que puede tomar n valores. Según el valor

que la expresión tenga en cada momento se ejecutan las acciones correspondientes al valor.

Figura 18: Diagrama de Flujo de Decisión Múltiple

PSEUDOCÓDIGO:

Según sea <expresión>

```
<Valor1>: <acción1>  
<valor2>: <acción2> .....  
[<otro>: <acciones>]
```

fin según

Ejemplo en pseudocódigo:

INICIO

```
ENTERO posicionDeLlegada = 3  
SEGUN SEA posicionDeLlegada  
    1: entregar medalla de oro  
    2: entregar medalla de plata  
    3: entregar medalla de bronce  
    otro: entregar mención especial
```

FIN

Es importante mencionar que la estructura anterior puede ser escrita usando los condicionales vistos anteriormente de la siguiente forma:

INICIO

```
ENTERO posicionDeLlegada = 3  
SI (posicionDeLlegada = 1)  
ENTONCES:  
    entregar medalla de oro  
SINO:  
    SI (posicionDeLlegada = 2)
```

ENTONCES:

entregar medalla de plata

SINO:

SI (posicionDeLlegada = 3)

ENTONCES:

entregar medalla de bronce

SINO:

entregar mención especial

FIN_SI

FIN_SI

FIN_SI

FIN

Podemos ver que usar condiciones anidadas podemos resolver el mismo problema, pero la estructura resultante es mucho más compleja y difícil de modificar.

Estructura repetitiva o iterativa

Durante el proceso de creación de programas, es muy común, encontrarse con que una operación o conjunto de operaciones deben repetirse muchas veces. Para ello es importante conocer las estructuras de algoritmos que permiten repetir una o varias acciones, un número determinado de veces.

Las estructuras que repiten una secuencia de instrucciones un número determinado de veces se denominan **BUCLES o CICLOS**. Y cada repetición del bucle se llama **iteración**.

Todo bucle tiene que llevar asociada una **condición**, que es la que va a determinar cuándo se repite el bucle y cuando deja de repetirse.

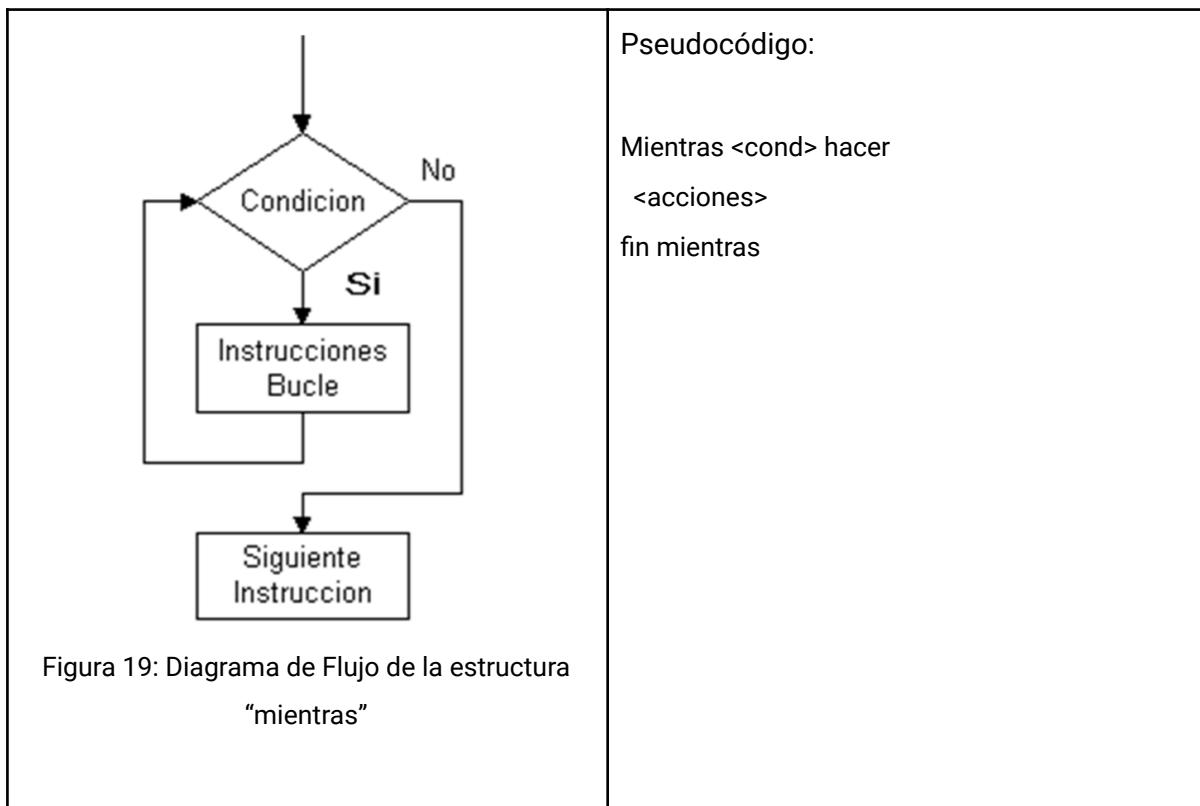
Un bucle se denomina también lazo o **loop**. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalización del bucle no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores principiantes.

Hay distintos tipos de bucles:

- Mientras, en inglés: **While**
- Hacer Mientras, en inglés: **Do While**.
- Para, en inglés: **For**

Estructura mientras (while, en inglés)

Esta estructura repetitiva “mientras”, es en la que el cuerpo del bucle se repite siempre que se cumpla una determinada condición.



Ejemplo:

INICIO

```
BOOLEANO tanqueLleno = falso
MIENTRAS (tanqueLleno == falso)
  HACER:
```

```

    llenar tanque
  FIN_MIENTRAS
  // el tanque ya está lleno :)

FIN
  
```

Estructura hacer-mientras (do while, en inglés)

Esta estructura es muy similar a la anterior, sólo que a diferencia del while el contenido del bucle se ejecuta siempre al menos una vez, ya que la evaluación de la condición se encuentra al final. De esta forma garantizamos que las acciones dentro de este bucle sean llevadas a cabo, aunque sea una vez independientemente del valor de la condición.

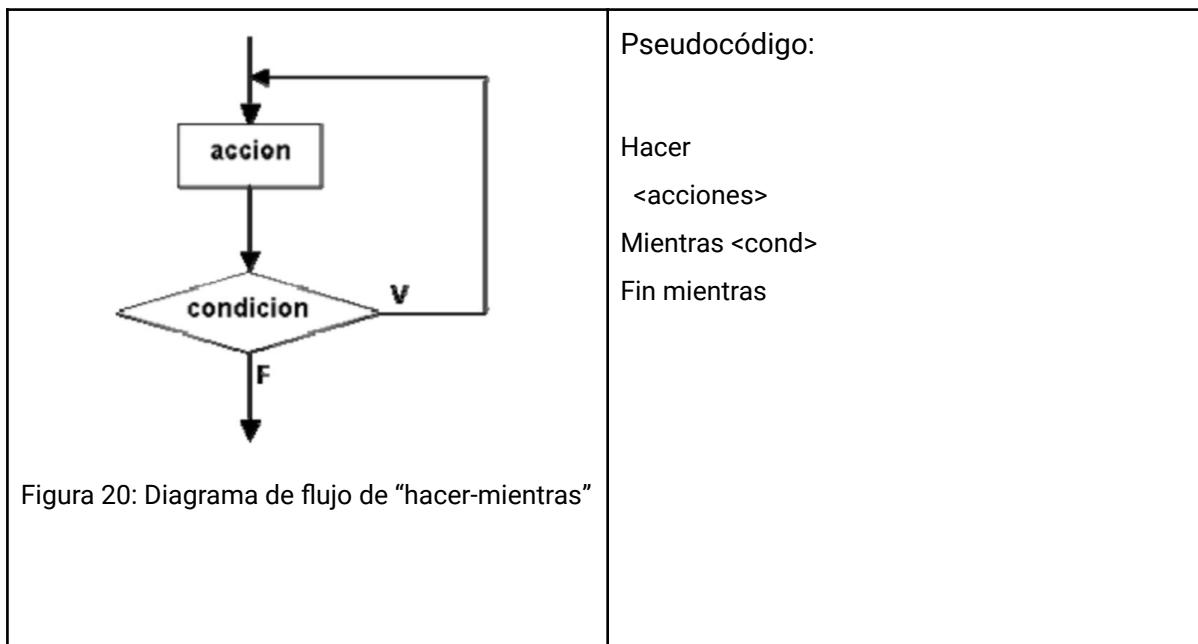


Figura 20: Diagrama de flujo de “hacer-mientras”

Ejemplo:

INICIO

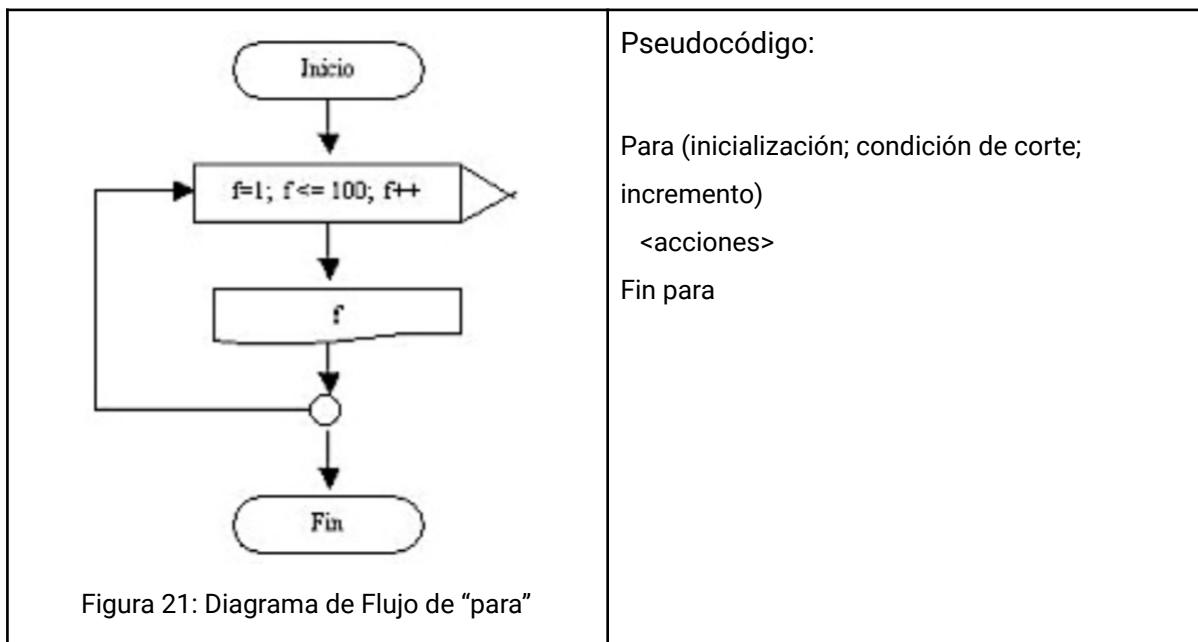
```

  BOLEANO llegadaColectivo=false;
  HACER: esperar en la parada
  MIENTRAS (llegadaColectivo == false)
    FIN_MIENTRAS
  
```

FIN

Estructura para (for, en inglés)

La estructura for es un poco más compleja que las anteriores y nos permite ejecutar un conjunto de acciones para cada elemento de una lista, o para cada paso de un conjunto de elementos. Su implementación depende del lenguaje de programación, pero en términos generales podemos identificar tres componentes: la **inicialización**, la **condición** de corte y el **incremento**.



Ejemplo:

INICIO

PARA (ENTERO RUEDA = 1; RUEDA <= 4; RUEDA++)

 inflar_rueda (RUEDA)

FIN_PARA

FIN

La ejecución del pseudocódigo anterior dará como resultado las siguientes llamadas a la función inflar():

1. inflar_rueda (1)

2. inflar_rueda (2)

3. inflar_rueda (3)

4. inflar_rueda (4)

Luego de esto podríamos suponer que hemos inflado las 4 cubiertas del auto y estamos listos para seguir viaje.

Recursividad

La recursividad es un elemento muy importante en la solución de algunos problemas de computación. Por definición, un algoritmo recursivo es aquel que utiliza una parte de sí mismo como solución al problema. La otra parte generalmente es la solución trivial, es decir, aquella cuya solución será siempre conocida, es muy fácil de calcular, o es parte de la definición del problema a resolver. Dicha solución sirve como referencia y además permite que el algoritmo tenga una cantidad finita de pasos.

La implementación de estos algoritmos se realiza generalmente en conjunto con una estructura de datos, la pila, en la cual se van almacenando los resultados parciales de cada recursión.

Un ejemplo es el cálculo de factorial de un número, se puede definir el factorial de un número entero positivo x como sigue:

$$x! = x * (x-1) * (x-2) \dots * 3 * 2 * 1$$

donde “!” indica la operación unaria de factorial. Por ejemplo, para calcular el factorial de 5, tenemos:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Definimos, además:

$$1! = 1 \text{ y } 0! = 1$$

Sin embargo, podemos observar que la definición del factorial de un número x , puede expresarse, a su vez, a través del factorial de otro número:

$$x! = x * (x-1)!$$

Es decir, para conocer el factorial de x basta con conocer el factorial de $x-1$ y multiplicarlo por x . Para conocer el factorial de $x-1$ basta con conocer el factorial de $x-2$, y multiplicarlo por $x-1$. Este proceso se realiza recursivamente, hasta llegar a la solución trivial, donde necesitamos el factorial de 1, el cual es 1.

Lo importante a notar en la igualdad anterior es que expresa un proceso recursivo, donde definimos una operación en términos de sí misma.

El pseudocódigo queda así:

Factorial (x)

```
SI (x == 1 O x == 0) ENTONCES:  
    DEVOLVER 1  
SINO:  
    DEVOLVER x * Factorial (x-1)  
FIN_SI
```

Ventajas

- Algunos problemas son esencialmente recursivos, por lo cual su implementación se facilita mediante un algoritmo de naturaleza recursiva, sin tener que cambiarlo a un método iterativo, por ejemplo.
- En algunas ocasiones el código de un algoritmo recursivo es muy pequeño

Desventajas

- Puede llegar a utilizar grandes cantidades de memoria en un instante, pues implementa una pila cuyo tamaño crece linealmente con el número de recursiones necesarias en el algoritmo. Si los datos en cada paso son muy grandes, podemos requerir grandes cantidades de memoria lo que a veces puede agotar la memoria de la computadora donde está corriendo el programa.

Estructuras de Datos: Pilas, Colas y Listas

En esta sección veremos estructuras de datos que nos permiten colecccionar elementos.

Para poder agregar u obtener elementos de estas estructuras de datos tenemos dos operaciones básicas:

- COLOCAR
- OBTENER.

Dependiendo de cómo se realicen las operaciones sobre los elementos de la colección es que definimos pilas, colas y listas.

Listas

Una lista (en inglés array) es una **secuencia de datos**. Los datos se llaman elementos del array y se numeran consecutivamente 0, 1, 2, 3, etc. En una lista los datos deben ser del mismo tipo. Es importante destacar que la mayoría de las estructuras de datos en los lenguajes de programación son zero-based, es decir que el primer elemento siempre tendrá asignado el número de orden 0, lo que significa que la cantidad de elementos total será igual al número del último elemento más 1. El tipo de elementos almacenados en la lista puede ser cualquier tipo de dato. Normalmente la lista se utiliza para almacenar tipos de datos, tales como cadenas de texto, números enteros o decimales.

Una lista puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de asientos que tiene un colectivo de larga distancia. Cada ítem de una lista se denomina elemento. Una lista tiene definido una longitud, que indica la cantidad de elementos que contiene la misma. Por ejemplo, si tenemos una lista que contiene los meses del año, entonces dicha lista tiene en total 12 elementos, donde el primer elemento “Enero” tiene el número de orden 0 y “Diciembre” el número 11.

Los elementos de una lista se enumeran consecutivamente 0, 1, 2, 3, 4, 5, etc. Estos números se denominan valores **índices** o **subíndice** de la lista.

Los índices o subíndices de una lista son números que sirven para identificar únicamente la posición de cada elemento dentro de la lista. Entonces, si uno quiere acceder a un elemento determinado de la lista, conociendo su posición, es decir su índice, se puede obtener el elemento deseado fácilmente.

Podemos representar una lista de la siguiente forma: Si consideramos una lista de longitud 6:

elemento 1	elemento 2	elemento 3	elemento 4	elemento 5	elemento 6
índice: 0	índice: 1	índice: 2	índice: 3	índice: 4	índice: 5

Ejemplo, volviendo al ejemplo planteado en el párrafo superior, se muestra a continuación una lista que contiene todos los meses del año:

Nombre de la Lista: **mesesDelAño**

Longitud de la Lista: **12**

Tipo de Datos: **String**

Ene	Feb	Mar	Abr	May	Jun	Jul	Ago	Sep	Oct	Nov	Dic
0	1	2	3	4	5	6	7	8	9	10	11

Si quisieramos acceder a un elemento determinado de la lista, simplemente debemos conocer cuál es la posición del elemento deseado.

Ejemplo:

INICIO

LISTA **mesesDelAño**

OBTENER(**mesesDelAño**, 11) // esto nos devuelve "diciembre"

OBTENER(**mesesDelAño**, 0) // esto nos devuelve "enero"

```
OBTENER(mesesDelAño, 7) // esto nos devuelve "agosto" OBTENER(mesesDelAño,  
12) // error: no existe el elemento 12
```

FIN

Si quisiéramos asignar un valor a una posición determinada de la lista, necesitamos conocer por un lado la posición que queremos asignar, y el elemento que vamos a asignar a dicha posición:

INICIO

```
LISTA mesesDelAño
```

```
COLOCAR(mesesDelAño, 10, "noviembre") // "noviembre", en la posición 10  
COLOCAR(mesesDelAño, 0, "enero") // esto asigna "enero", en la posición 0  
COLOCAR(mesesDelAño, 3, "abril") // esto asigna "abril", en la posición 3
```

FIN

Estructuras de pilas y colas no forman parte del dictado de este curso. Para más información dirígete a [https://es.wikipedia.org/wiki/Pila_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)) o [https://es.wikipedia.org/wiki/Cola_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)).

Javascript

¿Qué es JavaScript?

JavaScript es un lenguaje de scripting multiplataforma y orientado a objetos. Es un lenguaje pequeño y liviano. Dentro de un ambiente de host, JavaScript puede conectarse a los objetos de su ambiente y proporcionar control programático sobre ellos.

JavaScript contiene una librería estándar de objetos, tales como `Array`, `Date`, y `Math`, y un conjunto central de elementos del lenguaje, tales como operadores, estructuras de control, y sentencias. El núcleo de JavaScript puede extenderse para varios propósitos, complementándolo con objetos adicionales, por ejemplo:

- **Client-side JavaScript** extiende el núcleo del lenguaje proporcionando objetos para controlar un navegador y su modelo de objetos (o DOM, por las iniciales de Document Object Model). Por ejemplo, las extensiones del lado del cliente permiten que una aplicación coloque elementos en un formulario HTML y responda a eventos del usuario, tales como clicks del ratón, ingreso de datos al formulario y navegación de páginas.
- **Server-side JavaScript** extiende el núcleo del lenguaje proporcionando objetos relevantes a la ejecución de JavaScript en un servidor. Por ejemplo, las extensiones del lado del servidor permiten que una aplicación se comunique con una base de datos, proporcionar continuidad de la información de una invocación de la aplicación a otra, o efectuar manipulación de archivos en un servidor.

¿Qué debo conocer previamente?

Si has llegado hasta aquí se supone que has adquirido los siguientes conocimientos básicos previamente:

- Un entendimiento general del Internet y de la World Wide Web (WWW).

Buen conocimiento práctico del lenguaje de marcado HTML (HyperText Markup Language) (HTML).

Historia

Nos encontramos en 1995. El invento de la web apenas tiene 4 años de vida y para conectarse a internet necesitamos hacerlo con un router módem de 56k que tiene una melodía entrañable. Si nos conectamos a la red nos quedamos sin teléfono en casa. Y la conexión se cobra a coste de llamada telefónica. La gente que se conecta a Internet es mucha teniendo en cuenta las dificultades tecnológicas del momento.

Vivimos en una década dónde el monopolio de Microsoft en los ordenadores personales es una evidencia pero en que una pequeña empresa, Netscape, surge como uno de los nuevos gigantes de la red.

Para Microsoft, que la gente pudiese navegar por internet sin tener que utilizar windows significaba el principio del fin de su hegemonía. No se lo podía permitir.

De modo que era necesario hacer todo lo posible por imponer una solución propia a los usuarios para recuperar el control. Microsoft se puso a trabajar en una copia de Netscape.

Netscape gozaba de la simpatía de los usuarios. Era una herramienta de vanguardia que funcionaba muy bien para navegar. Era una herramienta de pago.

Y por su contra, Microsoft tenía la fuerza de los grandes acuerdos comerciales con la fabricantes de ordenadores, una plantilla muy talentosa y enorme y el control total de Windows. Si Netscape sacaba una nueva innovación, Microsoft hacía todo lo posible por copiarla. Aunque fuese tarde.

Y en un inteligente movimiento para asegurarse el dominio del Mercado, con windows 95 Microsoft incluyó su navegador Explorer (una copia de Netscape) gratuito junto con su sistema operativo.

Más adelante esto desencadenaría en una denuncia antimonopolio contra Microsoft que seguramente a muchos nos sonará.

En Netscape trabajaba un joven talento llamado **Brendan Eich (CEO actual de Mozilla)**. A Brendan le encomendaron crear un nuevo lenguaje de programación en 10 días porque tenía que salir con la nueva versión del navegador.

El objetivo de dicho lenguaje era que las páginas web pudieran aprovechar el poder de procesamiento de los ordenadores para poder hacer la navegación por Internet más rápida.

En las páginas convencionales de la época, para hacer una simple suma se tenía que cargar un documento, hacer la operación matemática en el servidor y enviarla a un nuevo documento que se tenía que cargar de nuevo. A 56k de velocidad de la red para la mejor de las conexiones , podemos imaginar la lentitud del proceso.

Por poner un ejemplo, Javascript aceleraba enormemente la validación de datos con formularios por internet.

En sus inicios este lenguaje se bautizó como **Mocha**, y posteriormente se le cambiaría el nombre a **Livescript**. Nacía así una nueva tecnología de Internet.

En un intento por gozar de más peso en el mercado, en 1995 se inició un movimiento de marketing por el que se quería asociar Livescript con Java. Hubo un acuerdo entre Sun Microsystems y Netscape por el que Livescript pasó a llamarse Javascript. Pero lo cierto es que Javascript y Java no tienen nada que ver salvo en algunas aspectos por lo que son lenguajes distintos.

Considerando el impacto de Javascript (JS) , Microsoft decidió no quedarse a un lado e hizo una copia. Creó su propio lenguaje denominado JScript 1.0. Y lo implementó en su navegador Explorer desde la versión 3.0 y en su servidor web, Internet Information Server.

Hoy en día todos los profesionales del sector estamos de acuerdo en considerar Javascript como uno de los pilares de la web. Sin embargo durante muchos años fueron pocos los visionarios que le otorgaban este papel principal a este lenguaje. En realidad, no era considerado un lenguaje serio... De 1997 a 2007 hubo muy pocos cambios en el standard de ECMAscript.

Fueron empresas como Yahoo y Google con las appwebs de Gmail, Google Maps, ... las que apostaron fuerte por este lenguaje y supieron ver todo su potencial. Podríamos decir que hasta 2004 el lenguaje no empezó a despegar. Pero si tuvieramos que poner un momento clave en el desarrollo de Javascript este sería 2006 con el lanzamiento de Jquery(una librería muy usada hasta hace unos años para acceder al DOM).

Jquery es un marco de trabajo que unificaba el código de todos los navegadores webs para garantizar que una misma instrucción de código funcionaba correctamente en todas partes. A partir de aquí las librerías se fueron sucediendo. node.js... y un momento de máximo esplendor lo encontramos con la nueva etiqueta <canva> de html5 que combinada con Javascript permite hacer auténticas maravillas.

Actualmente el lenguaje permite programar robots, aplicaciones web, aplicaciones móviles, aplicaciones de escritorio ... y dar respuesta a un sinfín de necesidades impensables hace 15 años.

¡Hablar de Javascript es hablar del presente y el futuro de la web!

JavaScript y Java

JavaScript y Java son similares en algunos aspectos, pero fundamentalmente diferentes en otros. El lenguaje JavaScript se parece a Java pero no tiene el tipo estático (static) de Java, ni tiene un chequeo de tipos fuerte. JavaScript usa la mayoría de la sintaxis de expresiones de Java, convenciones de nombrado, y las construcciones básicas de control de flujo, razón por la cual se le cambió el nombre del original LiveScript a JavaScript.

En contraste con el sistema de clases construidas por declaraciones que se usa en tiempo de compilación de Java, JavaScript soporta un sistema de tiempo de ejecución basado en un pequeño número de tipos de datos que representan valores numéricos, lógicos, y de cadena de caracteres (string). JavaScript tiene un modelo de objetos basado en prototipos en lugar del modelo de objetos basado en clases, que es más común. El modelo basado en prototipo proporciona herencia dinámica; esto es, que lo que se hereda puede variar entre objetos individuales. JavaScript también soporta funciones sin ningún requerimiento declarativo especial. Las funciones pueden ser propiedades de los objetos, ejecutándose como métodos levemente tipados.

Comparado con Java, JavaScript es un lenguaje muy libre de forma. No hay que declarar todas las variables, clases, y métodos. No hay que preocuparse de si los métodos son públicos, privados, o protegidos, y no hay que implementar interfaces. Las variables, parámetros, y retornos de funciones no tienen que declararse explícitamente de un tipo dado.

Java es un lenguaje de programación basado en clases, diseñado para lograr seguridad de tipos, y ejecución rápida. Seguridad de tipos significa, por ejemplo, que no es posible hacer una conversión de tipos a un Integer de Java para obtener una referencia a un objeto, o acceder a memoria protegida mediante la alteración del bytecode de una clase. El modelo basado en clases de Java implica en que los programas consisten exclusivamente en clases y sus métodos. La herencia de clases de Java y el tipado fuerte generalmente requiere que jerarquías de objetos fuertemente acopladas. Estos requerimientos hacen que la programación en Java sea más compleja que la programación en JavaScript.

En contraposición, JavaScript proviene en espíritu de una línea de lenguajes de programación más pequeños, con tipado dinámico, como HyperTalk, y dBASE. Estos lenguajes de scripting hacen accesibles las herramientas de programación a audiencias mucho más amplias. Esto se debe a su sintaxis más indulgente, funcionalidad especializada ya incluida, y mínimos requisitos para la creación de objetos.

ECMAScript

En el año de 1997 se crea un comité (TC39) en la ECMA para estandarizar JavaScript. A partir de entonces, los estándares de JavaScript se rigen como ECMAScript. No solo JavaScript se basa en el lenguaje ECMAScript, existen otros como JScript y ActionScript 3 que también lo hacen. Haciendo una analogía, diremos que ECMAScript es el lenguaje y JavaScript, JScript y ActionScript 3 son dialectos de este lenguaje, siendo JavaScript su dialecto más conocido y utilizado.

Debido a esto no existen versiones propias de JavaScript, sino de su estándar contenedor ECMAScript, que desde el 2015 se encuentra en su versión número 6, esta trae cambios en la sintaxis del lenguaje importantes que lo veremos más adelante.

Ejecutando código JavaScript en el navegador

Existen dos formas de ejecutar código JavaScript en los navegadores: a través de las herramientas de desarrollador que trae el navegador y creando un archivo HTML que incluya código JavaScript.

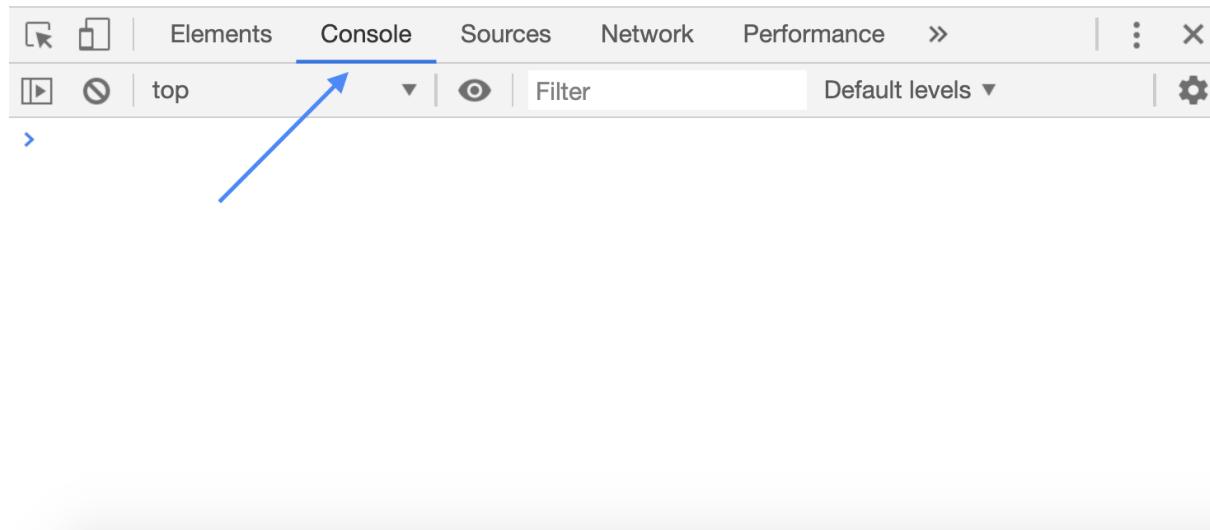
A través de las herramientas de desarrollador

Las herramientas de desarrollador (en inglés developer tools) son un conjunto de herramientas integradas al navegador que utilizan los Desarrolladores Front End para analizar, depurar y mejorar su código.

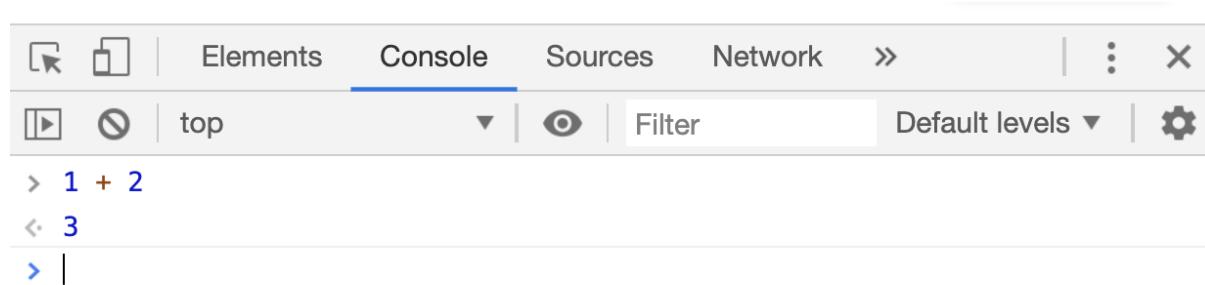
La forma más fácil de abrir las herramientas de desarrollador en cualquier navegador es hacer click en cualquier parte de la página y seleccionar la opción "Inspeccionar Elemento" en el menú desplegable que aparece.

También existe un atajo del teclado para abrir y cerrar las herramientas de desarrollador. El atajo para la mayoría de navegadores en Mac es **Alt + Command + I**. Para PC es **Ctrl + Shift + I**.

Una de las herramientas que incluyen las herramientas de desarrollador es la Consola, que la puedes abrir haciendo click en la pestaña "Consola" (o en Inglés "Console") como se muestra en la siguiente imagen.



En la **Consola** podemos escribir una expresión de JavaScript, oprimir **Enter**, y veremos el resultado de esa expresión en la siguiente línea. Por ejemplo, escribe `1 + 2` y oprime Enter. Deberás ver el número 3 en la siguiente línea como se muestra en la siguiente imagen.

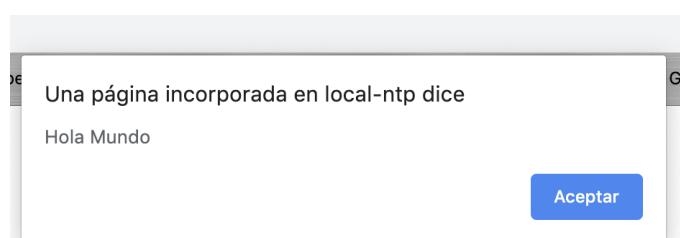


A través de un documento HTML

La otra forma de ejecutar código JavaScript en el navegador es dentro de un documento HTML. Crea un archivo llamado **index.html** y pega el siguiente contenido como se muestra a continuación.

```
<!DOCTYPE html>
<html>
<head>
<title>Ejemplo JavaScript</title>
<script>
alert("Hola Mundo");
</script>
</head>
<body>
</body>
</html>
```

Ábrelo con tu navegador preferido. Deberías ver un mensaje de alerta con el texto "**Hola Mundo**".



Aunque insertar el código directamente dentro del HTML funciona, se considera **una mala práctica**. Crea un nuevo archivo llamado **app.js** en la misma carpeta donde se encuentre **index.html** y pega el siguiente contenido:

```
alert("Hola Rolling");
```

Ahora modifica **index.html** con el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo JavaScript</title>
  <script src="app.js"></script>
</head>
<body>

</body>
</html>
```

Ábrelo nuevamente con un navegador o refresca la página si ya lo tenías abierto. Deberías ver una alerta pero ahora con el texto "**Hola Rolling**".

Si llegaste hasta aquí. Haz logrado construir tu primera aplicación con Javascript!.

Vemos que para poder escribir código javascript necesitamos usar la etiqueta **script** la cual dentro de ella escribimos nuestro código. También si usamos el atributo **src** de esta etiqueta

podemos enlazar un archivo externo con extensión **js** donde colocaremos todo nuestro código javascript.

Una tercera forma de ejecutar código javascript es colocarlo directamente dentro de elementos html como vemos a continuación:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo JavaScript</title>
  </head>
  <body>
    <button onclick="alert('Hola Rolling')">Mostrar Mensaje</button>
  </body>
</html>
```

Al hacer click sobre sobre el botón llamado **Mostrar Mensaje**, se mostrará el mensaje **Hola Rolling**

Comentarios

Los comentarios se utilizan para documentar o aclarar nuestro código y son ignorados al ejecutar el archivo. En JavaScript se utilizan los caracteres **//** para crear un comentario de una línea. Por ejemplo:

```
// este es un comentario de una linea
console.log("Hola Mundo");
console.log("Esto está muy copado"); // este es otro comentario
```

También puedes crear comentarios de múltiples líneas encerrando el texto entre **/*** y ***/**. Por ejemplo:

```
/*
este es un comentario
de varias líneas
```

```
console.log("esto no es ignorado al ejecutar el archivo")
*/
console.log("Hola Mundo");
console.log("Esto está muy bueno");
```

Fíjate que la última línea del comentario es código JavaScript válido. Sin embargo, ese código **no se ejecuta porque está como comentario**.

Tipos y operadores

En esta sección vamos a hablar sobre cadenas de texto, números, booleanos (verdadero o falso), valor no definidos y nulos, que son tipos de datos que más se usan en JavaScript, y cómo realizar algunas operaciones con ellos. Pero antes en la siguiente tabla se muestran todos los tipos de datos que existen en Javascript.

Tipo	Primitivo	Ejemplo
String	Si	"Hola Mundo"
Number	Si	42 o 3.14159
Boolean	Si	Verdadero o Falso
undefined	Si	representa el valor que toma una variable que no ha sido inicializada específicamente
null	Si	representa un valor nulo o "vacío"
Symbol	Si	nuevo en ECMAScript 6
Object	No	representan datos complejos

Empecemos con las cadenas de texto.

Cadenas de texto (Strings)

Una cadena de texto es un conjunto de caracteres encerrados entre comillas simples ('') o dobles (""). Por ejemplo:

```
"Texto entre comillas dobles"
```

```
'Texto entre comillas simples'
```

Aunque parece fácil, existen tres errores comunes al definir una cadena de texto para que los tengas en cuenta e intentes evitarlos:

- Olvidarse de la comilla de cierre. Por ejemplo:

```
"Hola mundo
```

- Cerrar la cadena con la comilla incorrecta (p.e. abrir la cadena con comilla doble y cerrarla con comilla sencilla). Por ejemplo:

```
"Hola mundo'
```

- Insertar el mismo tipo de comillas dentro de la cadena de texto. Por ejemplo:

```
"Y él dijo: "Hola Mundo""
```

```
'Hol'a mundo'
```

Para solucionar el último error (punto 3) podemos encerrar la primera cadena entre comillas simples y la segunda entre comillas dobles para que funcione:

```
"Hol'a mundo"
```

```
'Y él dijo: "Hola mundo"'
```

Importante: Lo importante es que el texto no contenga la comilla que se utilizó para definir la cadena.

Pero ¿qué pasa si tenemos un texto con comillas simples y dobles? En ese caso tendríamos que utilizar el carácter de escape \ como en el siguiente ejemplo:

'Y \'él dijo\'': "Hola mundo"

o

"Y 'él dijo': \"Hola mundo\""

En el primer ejemplo escapamos con \ las comillas simples porque con esas fue que encerramos el texto, mientras que en el segundo ejemplo escapamos las comillas dobles porque con esas fue que encerramos el texto.

Imprimir una cadena de texto

Imprimir es un término que se usa mucho en los lenguajes de programación cuando queremos mostrar información al usuario por algún dispositivo de salida del computador. Lo más común es querramos mostrar la información en la pantalla o monitor del usuario. Para imprimir una cadena de texto en la línea de comandos (o en la consola del navegador) utilizamos la herramienta de javascript denominada como **console.log**:

```
console.log('Y \'él dijo\'': "Hola mundo");
```

Si guardas esa línea en un archivo llamado strings.js, lo vinculas a un archivo html y lo ejecutas en google chrome, el resultado debería ser el siguiente:

```
> Y 'él dijo': "Hola mundo"
```

Ojo!. El texto anterior se visualiza en la consola del navegador y no en el viewport. Más adelante aprendemos cómo manipular elementos html desde javascript para colocar allí la información resultante de los datos devueltos de algún algoritmo.

Concatenar cadenas

Es posible unir cadenas de texto con el operador +. Por ejemplo, abre la consola de google Chrome y ejecuta lo siguiente:

```
"Hola" + "Mundo" + "Cómo" + "Estás"
```

Deberías ver algo como esto:

```
> "Hola" + "Mundo" + "Cómo" + "Estás"  
> HolaMundoCómoEstás
```

Vemos que las palabras no se separan con espacio automáticamente, tenemos que agregar los espacios explícitamente:

```
> console.log("Hola " + "Mundo " + "Cómo " + "Estás");  
> Hola Mundo Cómo Estás
```

Debemos tener cuidado al concatenar cadenas y números.

En este momento la concatenación de cadenas no es muy útil porque hubiésemos podido escribir todo el texto "**Hola Mundo Cómo Estás**" dentro de una sola cadena, pero a medida que veamos otros conceptos se va a volver cada vez más importante.

Números (Number)

Crea un nuevo archivo llamado numbers.js con el siguiente código:

```
console.log(1 + 2)  
console.log(3 * 4 + 5)  
console.log(8 / 2)
```

Vincula el archivo creado anteriormente a un documento html y ejecutalo en google chrome. El resultado que deberías ver, es el siguiente:

```
3  
17  
4
```

Echemos un vistazo en la segunda línea del ejemplo anterior. JavaScript sigue el mismo orden de precedencia en las operaciones matemáticas, tal y como nos enseñaron en la escuela. Por lo tanto la multiplicación se ejecuta primero que la suma. Podemos cambiar

este comportamiento usando los paréntesis. Por ejemplo, cambia la operación de la segunda línea por `3 * (4 + 5)`. El resultado ahora debería ser `27`.

A diferencia de las cadenas de texto, los números no se encierran entre comillas de ningún tipo (de lo contrario JavaScript los trata como texto y no como números). Por ejemplo, abre la consola de google chrome y escribe `"1" + "2"`. El resultado ya no es 3, es la cadena de texto `"12"`:

```
> "1" + "2"  
> '12'
```

Debes tener cuidado al concatenar cadenas y hacer sumas, porque los dos utilizan el mismo operador `+`. Por ejemplo, intenta lo siguiente en la consola del navegador:

```
> "1 + 2 es " + 1 + 2  
> '1 + 2 es 12'  
> "1 + 2 es " + (1 + 2)  
> '1 + 2 es 3'
```

En el primer ejemplo JavaScript toma la cadena `"1 + 2 es "` y la concatena con `"1"`, luego concatena el `"2"`, y el resultado es `"1 + 2 es 12"`.

En el segundo ejemplo JavaScript realiza primero la operación `(1 + 2)`, que da `3`, y luego concatena la cadena `"1 + 2 es "` con ese `3`, por eso es que el resultado es ahora `"1 + 2 es 3"`.

Valores y expresiones booleanas (Boolean)

Existen dos valores booleanos en programación: **verdadero (true)** y **falso (false)**. Abre la consola de navegador, escribe `true` y oprime Enter, después escribe `false` y oprime Enter. El resultado debe ser el siguiente:

```
> true  
true  
> false  
false
```

También es posible escribir expresiones que evalúen a true o false. Escribe ahora las siguientes expresiones en la consola:

```
5 > 3
5 >= 3
4 < 4
4 <= 4
2 === 2
2 !== 2
```

El resultado debería ser el siguiente:

```
> 5 > 3
true
> 5 >= 3
true
> 4 < 4
false
> 4 <= 4
true
> 2 === 2
true
> 2 !== 2
false
> "ruby" === "javascript"
false
> "ruby" !== "javascript"
true
```

A los operadores `<`, `>`, `<=`, `>=`, `==`, `!=` se les llama **operadores relacionales** y se utilizan para crear expresiones que se evalúan a un valor booleano: **verdadero (true)** o **falso (false)**. Las expresiones lógicas sólo contienen 1 valor como resultado (true o false). Se utiliza normalmente en la programación, estadística, electrónica, matemáticas (Álgebra booleana), etc.

En JavaScript existe el operador == y el === (así como el != y !==). Veamos dos ejemplos para explicar la diferencia:

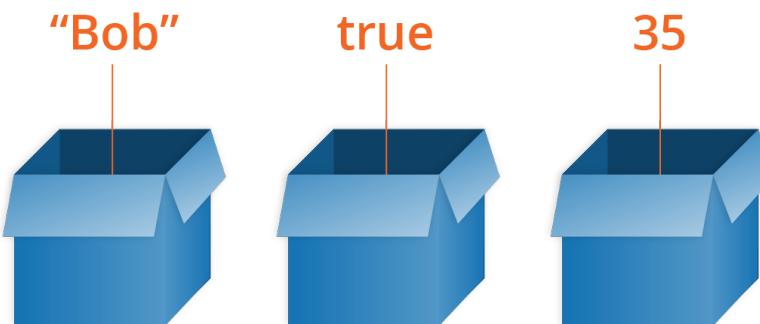
```
> 1 == "1"  
true  
> 1 === "1"  
false
```

En los dos ejemplos estamos comparando el número 1 con la cadena de texto "1". En la mayoría de lenguajes esto sería **false** porque son tipos diferentes, estamos comparando un número con una cadena de texto. Pero el == realiza una conversión de tipos primero y después evalúa la igualdad.

Nota: Hoy el uso de == y el != se considera mala práctica y de ahora en adelante utilizaremos el === y !== para hacer comparaciones.

Variables

Las variables como vimos en secciones anteriores son uno de los conceptos básicos de la programación y nos permiten almacenar información temporal que podemos usar más adelante en nuestros programas.



Para entenderlo mejor, podemos ver a las variables como cajitas donde dentro de ellas almacenados cualquier tipo de información.

Para profundizar más en el tema crea un archivo llamado variables.js y agrega lo siguiente:

```
var name = "Juan"; // cámbialo por tu nombre  
console.log("Hola " + name);
```

En este ejemplo estamos definiendo una variable con nombre **name** y le asignamos el valor "**Juan**" (o el valor que le hayas asignado). En la siguiente línea estamos utilizando concatenación de cadenas para mostrar la cadena de texto "**Hola** " seguido del valor que tenga en ese momento la variable **name**.

Declaraciones

Hay tres tipos de declaraciones de variables en JavaScript.

var

Declara una variable, inicializándola opcionalmente a un valor. Es la forma tradicional que se venía usando para declarar una variable.

let

Declara una variable local en un bloque de ámbito, inicializándola opcionalmente a un valor. Es la nueva forma de declarar variables. Se recomienda usar esta forma

const

Declara una constante de sólo lectura en un bloque de ámbito.

Creando variables

Las variables se crean con la palabra clave **var** o **let** o **const** (esto lo veremos más adelante) seguido del nombre de la variable. Opcionalmente, le puedes asignar un valor a la variable utilizando el operador de asignación = y el valor que le quieras dar. El punto y coma (;) al final es opcional pero se considera una buena práctica colocarlo.

El nombre de una variable debe comenzar con **\$**, **_** o una **letra**, y después puede contener letras, dígitos, **_** y **\$**. Ejemplos de nombres válidos de variables son:

```
var name,  
var $element  
let _trains.
```

```
let age
let country23
```

Por otro lado, ejemplos de nombres no válidos incluyen `443juan`, porque no puede empezar con un número, y `element&123`, porque el carácter & no es válido en el nombre.

Las palabras reservadas de JavaScript no se pueden usar como nombres de variables. Dicho tabla de palabras reservadas se adjuntará al final de este manual.

Como buena práctica se recomienda empezar las variables con una letra en minúscula y, si el nombre se compone de varias palabras, capitalizar cada palabra después de la primera. Por ejemplo `firstName` o `lastName`. Esta notación (forma de escribir nombres de variables) se denomina **Camel Case** y es una forma de escribir variables más común en los diferentes lenguajes de programación.

Los nombres de las variables diferencian mayúsculas y minúsculas (Ej. `firstname` es diferente a `firstName`), es decir, para Javascript son dos variables distintas.

Diferencia entre var y let

En este punto puedes estar pensando "¿por qué necesitamos dos palabras clave para definir las variables? ¿Por qué tener **var** y **let**?".

Las razones son un tanto históricas. Cuando Javascript fue creado, sólo existía `var`. Esto funciona bien en la mayoría de los casos, pero tiene algunos problemas en cómo funciona. Su diseño a veces puede ser confuso o molesto. Por eso, se creó `let` en versiones modernas de JavaScript, una nueva palabra clave que funciona de forma diferente a `var`, solucionando sus problemas. Más adelante profundizaremos en el tema.

La utilidad de las variables

Crea un archivo llamado `square.js` y escribe el siguiente código:

```
console.log("El perímetro de un cuadrado de lado 5 es " + (5 * 4));
console.log("El área de un cuadrado de lado 5 es " + (5 * 5));
```

Al ejecutarlo debería aparecer lo siguiente:

El perímetro de un cuadrado de lado 5 es 20

El área de un cuadrado de lado 5 es 25

El problema con este código es que si quisiéramos calcular el perímetro y el área de un cuadrado de lado 10, o 20, tendríamos que modificar ese valor en varias partes del código. Podemos mejorarlo utilizando una variable:

```
let side = 5;  
  
console.log("El perímetro de un cuadrado de lado " + side + " es " + (side * 4));  
console.log("El área de un cuadrado de lado " + side + " es " + (side * side));
```

Si ejecutas el código te debería dar el mismo resultado. La ventaja es que si quieres calcular el perímetro y el área de un cuadrado con otro tamaño sólo debes cambiar el valor de la variable `side`. Intenta con 10 (te debería dar 40 de perímetro y 100 de área) y después con 15.

Reasignar o cambiar el valor de las variables

Puedes cambiar el valor de una variable las veces que lo deseas. La forma de hacerlo es similar a la forma en que se declara una variable con un valor inicial, pero omitiendo la palabra `let`. Por ejemplo:

```
// asumiendo que name fue ya declarada  
name = "Nuevo valor";
```

Inténtalo. Abre la consola del navegador e ingresa lo siguiente:

```
> let name = "María" // declaramos name y le asignamos el valor "María"  
undefined  
> name          // verificamos el valor actual  
"María"  
> name = 123    // reasignamos el valor de name con el número 123  
123  
> name          // verificamos cuál es el valor actual de name  
123
```

Vemos que, como en este ejemplo, el nuevo valor que se le asigne a la variable no tiene que ser del mismo tipo del valor anterior.

También es posible reasignar el valor de una variable utilizando su valor anterior. Por ejemplo, intenta lo siguiente:

```
> let count = 1    // declaramos la variable con un valor inicial
undefined
> count      // verificamos cuál es el valor actual de count
1
> count = count + 1 // incrementamos en uno el valor actual de count
2
> count      // verificamos el valor actual de count
2
```

De hecho, incrementar el valor de una variable es tan común que existe un atajo para eso.

Asumiendo que sigues en la consola del navegador intenta lo siguiente:

```
> count++
3
> count++
4
```

Vemos que estos atajos normalmente se denominan **operadores unitarios**. Si tienes dudas sobre esto revisa las secciones anteriores donde presentamos una tabla sobre este tema.

Variables sin valor o undefined

En programación es muy común declarar una variable sin un valor, quizás porque más adelante vamos a pedirle el valor al usuario, o simplemente porque el valor se lo vamos a asignar después.

Una variable declarada sin un valor va a tener el valor de **undefined**.

Abre la consola del navegador e intenta lo siguiente:

```
> let name  
undefined  
> name  
undefined
```

Variables con null

null representa la ausencia de valor intencional, es decir que nosotros como programadores no sabemos qué valor va a contener una variable que declaramos, por lo que asignamos **null**. Este valor es muy común en muchos lenguajes de programación

Por ejemplo:

```
let name = null;  
console.log(name); // null  
console.log(typeof name); // advertencia! muestra "object"
```

El operador **typeof** indica cuál es el tipo de dato de una variable.

Cuidado con la última línea, tipo de una variable nula se muestra como «**object**», sin embargo es un bug, porque null no es un objeto.

Constantes

En palabras simples una constante es una variable que una vez definida su valor inicial no puede ser cambiado posteriormente. Es decir, una constante como su nombre lo indica mantiene un valor único durante todo el tiempo de vida del programa. Para crear una constante se usa la palabra clave **const**. La sintaxis del identificador de la constante es el mismo como para un identificador de variable: debe de empezar con una letra, guión bajo(_) o símbolo de dollar(\$) y puede contener alfabéticos, numéricos o guiones bajos.

Ejemplo:

```
const PI = 3.14;
```

Una constante no puede cambiar de valor mediante la asignación o volver a declararse mientras se ejecuta el programa.

Las reglas de ámbito para las constantes son las mismas que las de las variables let en un ámbito de bloque. Si la palabra clave const es omitida, el identificador se asume que representa una variable.

Ojo!. No puedes declarar una constante con el mismo nombre que una variable o una función (tema que veremos más adelante) en el mismo ámbito.

Un ejemplo de lo que **NO** se debería hacer:

```
let PI = 3.14
const PI = 3.14 //no se puede usar PI como nombre porque ya ha sido usado
```

¿Dónde y cuánto vive una variable?

Las variables se almacenan en una memoria especial del computador llamada **memoria RAM** y viven durante la ejecución del programa, es decir, desde el momento en que las defines hasta que tu programa termina de ejecutarse. Si abres la consola del navegador y defines una variable, esta vive hasta que cierres la pestaña.

La memoria RAM es una memoria de rápido acceso que está disponible mientras tu computador está encendido. El sistema operativo se encarga de administrar la memoria RAM y asignarle una porción a cada programa que se está ejecutando. Cuando el programa termina, el sistema operativo reclama esa memoria y "destruye" todas las variables que ese programa haya creado.

Ojo!. No confundir memoria RAM con Disco Rígido . Los dos son tipos de memorias pero para uso diferente.

Nota: Más adelante, cuando veamos funciones, aprenderemos que las variables y constantes tienen un alcance o ámbito y no todas sobreviven hasta que termina el programa.

Condicionales o estructuras de control

Hasta ahora hemos visto código que se ejecuta línea a línea, una detrás de otra. Pero a veces se hace necesario romper esa secuencia (**estructuras de control secuenciales**) y crear ramas que nos permitan tomar diferentes caminos en el código dependiendo de ciertas condiciones.

Por ejemplo, imagina cómo podríamos hacer un programa que nos diga si un número es mayor o menor a diez. Si es mayor a 10 debería imprimir una cosa, pero si es menor debería imprimir otra.

A este concepto se le conoce como condicionales y entra dentro de lo que son las **estructuras de control selectivas** que vimos en las secciones anteriores. La sintaxis en Javascript para estas estructuras es la siguiente:

Condicionales simples

```
if (<condición>) {  
    // código que se ejecuta si se cumple la condición  
}
```

La condición puede ser cualquier expresión que evalúa a **verdadero (true)** o **falso (false)**.

Crea un archivo llamado conditionals.js y agrega el siguiente contenido:

```
if (true) {  
    console.log("Hola Mundo");  
}
```

Nota: Las líneas que terminan con un corchete ({ o }) no se les agrega punto y coma (;).

Ejecuta el archivo desde la consola del navegador, deberías ver lo siguiente:

```
Hola Mundo
```

No importa cuántas veces ejecutes este archivo, el resultado siempre será el mismo.

Ahora probemos con falso (false) en vez de verdadero (true):

```
if (false) {  
    console.log("Hola Mundo");  
}
```

Ejecútalo. Esta vez nunca debería imprimir "**Hola mundo**", no importa cuantas veces lo ejecutes.

En vez de utilizar true o false como condición, podemos utilizar una expresión que evalúe a true o false.

```
if (1 == 1) {  
    console.log("Hola Mundo");  
}
```

El resultado al ejecutarlo debería ser:

Hola mundo

Prueba ahora con **1 == 2**, **1 < 6** y **8 < 6** en la condición y fíjate que tenga sentido.

Ahora que ya sabes cómo funciona los condicionales (muchos los llamamos los **ifs**) crea un programa en un archivo llamado number.js que imprima "**El número es menor a 10**" solo si el número que está almacenado en la variable num es menor a 10:

```
let num = 8;  
if (num < 10) {  
    console.log("El número es menor a 10");  
}
```

Si lo ejecutas te debería aparecer lo siguiente:

El número es menor a **10**

Ahora vamos a modificar el programa para que, además de imprimir "**El número es menor a 10**" si el número es menor a 10, también imprima "**El número es igual o mayor a 10**" si el

número es igual o mayor a 10. Podemos utilizar otro condicional debajo del que ya teníamos:

```
let num = 8;  
if (num < 10) {  
    console.log("El número es menor a 10");  
}  
if (num >= 10) {  
    console.log("El número es igual o mayor a 10");  
}
```

Ejecuta el programa e ingresa un número menor a 10, después un número mayor a 10, y por último 10. Verifica que el resultado sea el esperado.

Condicionales dobles o De lo contrario (else)

Lo único que necesitas para hacer condicionales es el if. Pero existen dos atajos que te van a permitir escribir código más corto.

El primer atajo es el **else**, que significa "**de lo contrario**" en Inglés. El else nos permite definir el código que se debe ejecutar si el if no se cumple, es decir si la condición evalúa a **falso**. La sintaxis es la siguiente:

```
if (<condición>) {  
    // código que se ejecuta si se cumple la condición  
} else {  
    // código que se ejecuta si NO se cumple la condición  
}
```

Podemos modificar el programa anterior, que nos dice si el número almacenado en la variable num es menor a 10, o si es mayor o igual, con un else.

```
let num = 8;  
if (num < 10) {  
    console.log("El número es menor a 10");  
} else {  
    console.log("El número es igual o mayor a 10");
```

```
}
```

Más corto y si lo ejecutas en la consola debería funcionar igual.

Condiciones múltiples - anidados

Ahora imagina que queremos modificar este programa para que en vez de imprimir "El número es igual o mayor a 10", imprima "El número es igual a 10" o "El número es mayor a 10" dependiendo si el número es igual 10 o mayor a 10 respectivamente.

En JavaScript (y en la mayoría de lenguajes de programación) es posible anidar condicionales, así que una posible solución sería la siguiente:

```
let num = 8;  
if (num < 10) {  
    console.log("El número es menor a 10");  
} else {  
    if (num > 10) {  
        console.log("El número es mayor a 10");  
    } else {  
        console.log("El número es igual a 10");  
    }  
}
```

Pruébalo con un número menor a 10, otro mayor a 10 y con 10. Te debería aparecer "El número es menor a 10", "El número es mayor a 10" y "El número es igual a 10" respectivamente.

Condicionales múltiples - De lo contrario, si (else if)

En general, es preferible no tener que anidar condicionales porque son difíciles de leer y entender. Otro atajo que nos ofrece JavaScript para los condicionales es el else if, que significa "**De lo contrario, si ...**" en Inglés. La sintaxis es la siguiente:

```
if (<primera condición>) {  
    // código que se ejecuta si <primera condición> se cumple  
} else if (<segunda condición>) {
```

```
// código si <primera condición> NO se cumple, pero <segunda condición> se cumple
} else if (<tercera condición>) {
    // código si <primera condición> y <segunda condición> NO se cumplen, pero <tercera
    // condición> sí se cumple
} else {
    // código si ninguna de las condiciones se cumple
}
```

Puedes definir tantos **else if** como deseas. El **else** es opcional.

Modifiquemos nuestro ejemplo anterior y en vez de utilizar condiciones anidadas, utilicemos **else if**:

```
let num = 8;
if (num < 10) {
    console.log("El número es menor a 10");
} else if (num > 10) {
    console.log("El número es mayor a 10");
} else {
    console.log("El número es igual a 10");
}
```

Lo más importante de entender en este código es que el programa sólo va a entrar a una de estas ramas. Por ningún motivo va a entrar a dos de ellas. Si la condición del primer if se cumple, el programa ejecuta el código que esté en ese bloque y después salta hasta después del else para continuar con el resto del programa o terminar.

Si la condición del primer if no se cumple, pero la del else if sí se cumple, el programa ejecuta el código de ese bloque y salta hasta después del else para continuar con el resto del programa o terminar.

Condicionales múltiples - según sea , el caso de

El condicional **switch** , se parece mucho y puede ser utilizado en lugar del **if** en ciertas ocasiones. Sin embargo, cuando quieras **evaluar el valor de una determinada expresión**

entre un **amplio número de posibilidades**, se prefiere utilizar el condicional `switch`. La razón es que enlazar muchos `if` y `else if` seguidos es difícil de leer y menos eficiente.

La sintaxis del condicional `switch` es la siguiente:

```
switch (expresión) {  
    case valor:  
        // bloque de código  
        break;  
    case valor:  
        // bloque de código  
        break;  
    default:  
        // bloque de código  
}
```

Cada **case**, se interpreta de la siguiente forma: "*si la expresión es igual al valor, ejecuta el bloque de código siguiente*". Si un **caso** es igual a la expresión, una vez que el código del caso se ha ejecutado, se hace uso del **break**. Esto hace que la ejecución salte fuera del código `switch` para que no se ejecute los casos que vienen a continuación. El **default** es como el **else** de los condicionales `if`, indica qué hacer si ninguno de los casos anteriores son iguales a la expresión.

Crea un archivo `switch.js` y coloca el siguiente código:

```
let estacion = prompt("¿Cuál es tu estación del año preferida?");  
switch (estacion) {  
    case "primavera":  
        // si la variable estacion contiene la cadena de texto "primavera"  
        // se ejecutará este bloque de código  
        console.log('la primavera');  
        break;  
    case "verano":  
        // si la variable estacion contiene la cadena de texto "verano"
```

```
// se ejecutará este bloque de código
console.log('el verano');

break;

case "otoño":
    // si la variable estacion contiene la cadena de texto "otoño"
    // se ejecutará este bloque de código
    console.log('el otoño');

    break;

case "invierno":
    // si la variable estacion contiene la cadena de texto "invierno"
    // se ejecutará este bloque de código
    console.log('el invierno');

    break;

default:
    // si la variable estacion no contiene ningún nombre válido
    // se ejecutará este bloque de código
    console.log('no es una estación del año');

}
```

En este ejemplo usamos la función **prompt** de javascript para solicitarle a través de una ventana que ingrese el nombre de la estación y se almacene ese valor en la variable **estación**.

Enlace el archivo anterior en un archivo html y ejecutalo en el navegador para probar.

Qué ocurre si no utilizamos break

Si retiramos la sentencia **break** de cada caso, obtendremos lo que se llama "falsos positivos". Se ejecutará todo el código debajo del primer caso coincidente, sin importar si los siguientes supuestos no coinciden con la expresión del switch.

Condiciones compuestas

Imaginemos que queremos escribir un programa que imprima "El número está entre 10 y 20" si el valor de una variable está efectivamente entre 10 y 20. ¿Cómo pensas que podríamos solucionar este problema?

Una opción es usar condiciones anidadas, de esta forma:

```
let num = 15;  
if (num >= 10) {  
    if (num <= 20) {  
        console.log("El número está entre 10 y 20");  
    }  
}
```

Sin embargo, como decíamos antes, leer condiciones anidadas es difícil y, en lo posible, es mejor evitarlas. En cambio, podemos utilizar los operadores lógicos **y** (`&&`) y **o** (`||`) para crear condiciones compuestas. El ejemplo anterior lo podemos mejorar con el operador **y**:

```
let num = 15;  
if (num >= 10 && num <= 20) {  
    console.log("El número está entre 10 y 20");  
}
```

Lo que estamos diciendo con este código es: **si el número es mayor o igual a 10 y menor o igual a 20** entonces imprima "El número está entre 10 y 20". Notamos que a cada lado del `&&` hay una expresión que evalúa a verdadero o falso: `num >= 10` y `num <= 20`.

Imaginemos ahora que necesitamos escribir un programa que imprima "Excelente elección" cuando el valor de una variable sea "rojo" o "negro" únicamente:

```
let color = "negro";  
if (color === "rojo" || color === "negro") {  
    console.log("Excelente elección");  
}
```

Forma de pensar como un programador

Vamos a jugar un juego llamado **Verdadero o Falso**. Yo digo una afirmación y tu debes responder si es verdadera o falsa. Trata de no mirar las respuestas debajo. Después comparas:

-
1. La Tierra gira alrededor del sol. (¿Verdadero o falso?)
 2. París es la capital de Estados Unidos.
 3. La Tierra gira alrededor del sol **y** los leones son animales.
 4. París es la capital de Estados Unidos **y** los leones son animales.
 5. La Tierra gira alrededor de Marte **y** los perros hablan Español.
 6. Los leones son animales **o** la Tierra gira alrededor del sol.
 7. París es la capital de Estados Unidos **o** los leones son animales.
 8. El planeta tierra gira alrededor de Marte **o** los perros hablan Español.

Las respuestas son las siguientes:

1. Verdadero.
2. Falso.
3. Verdadero.
4. Falso.
5. Falso.
6. Verdadero.
7. Verdadero.
8. Falso.

Cuando utilizamos **y**, las dos expresiones deben ser verdaderas para que el resultado sea verdadero. Cuando utilizamos **o**, cualquiera de las dos expresiones puede ser verdadera para que el resultado sea verdadero.

Evaluación de expresiones booleanas

Volvamos a jugar el juego, pero en vez de utilizar frases, utilicemos expresiones booleanas. Debes decidir si cada una de las siguientes expresiones es verdadera o falsa (**true** o **false**):

1. **true**
2. **false**
3. **1 < 1**
4. **2 != 3**
5. **1 < 1 && 2 != 3**

Copia y pega cada expresión en la consola del navegador para conocer las respuestas.

Analicemos la última expresión: `1 < 1 && 2 != 3`. ¿Cómo podemos saber si es verdadera o falsa?

El primer paso es evaluar cada lado de la expresión. `1 < 1` es `false` y `2 != 3` es `true`. Entonces quedaría:

`false && true`

Recuerda que para que una expresión con **y** (`&&`) sea verdadera, cada lado tiene que ser verdadero. Sin embargo, podemos hacer una tabla con todas las combinaciones entre verdadero y falso que podamos usar como referencia más adelante:

Expresión	Resultado
<code>true && true</code>	<code>true</code>
<code>true && false</code>	<code>false</code>
<code>false && true</code>	<code>false</code>
<code>false && false</code>	<code>true</code>

Vemos que el resultado solo es **true** cuando los dos lados del **&&** son **true**.

Hagamos lo mismo para el operador lógico **o** (`||`):

Expresión	Resultado
true true	true
true false	true
false true	true
false false	false

Con el operador **o** cualquiera de los lados puede ser **true** para que el resultado sea **true**.

A estas tablas se les conoce como **Tablas de Verdad**.

Hagamos algunos ejercicios. Decide si las siguientes expresiones evalúan a **true** o **false**.

Primero reemplaza cada lado del **&&** o el **||** y luego utiliza las tablas de verdad:

1. "hola" == "hola" && 1 < 2
2. true && 5 != 5
3. 1 == 1 || 2 != 1

Revisa tu respuesta evaluando cada expresión en la consola del navegador.

Podemos negar cualquier expresión booleana anteponiendo un signo de exclamación (!).

Por ejemplo:

1. **!true** es **false**
2. **!false** es **true**

De hecho, esa es la tabla de verdad de la **negación**. Intenta los siguientes ejercicios. Primero reemplaza lo que está entre paréntesis y luego aplica la tabla de verdad de la negación:

1. !(1 === 1)
2. !(2 <= 3)
3. !(true && 5 !== 5)

-
4. $!(1 < 1 \&& 2 != 3)$

El proceso para solucionar cualquier expresión booleana, sin importar qué tan compleja sea, es el siguiente:

1. Evalúa los operadores de igualdad ($<$, $>$, $==$, $!=$ etc).
2. Evalúa los **&&** y **||** que estén dentro de paréntesis.
3. Evalúa las negaciones ($!$).
4. Evalúa cualquier **&&** y **||** que falte.

Hagámoslo juntos. Intentemos evaluar la siguiente expresión booleana:

```
3 != 4 && !( "pedro" === "juan" || 26 > 10 )
```

1. Evaluar los operadores de igualdad:

```
true && !(false || true)
```

2. Evaluar los **&&** y **||** que estén dentro de paréntesis:

```
true && !true
```

3. Evaluar las negaciones:

```
true && false
```

4. Evaluar cualquier **&&** y **||** que falte:

```
false
```

Inténtalo tu. Decide si las siguientes expresiones evalúan a true o false:

1. $!(5 === 5) \&\& 8 != 8$
2. $("gut" === "ikk" \&\& 26 > 30) \|| ("gut" === "gut" \&\& 26 > 10)$
3. $!("testing" == "testing" \&\& !(5 > 8))$

Los condicionales en conjunto con las expresiones booleanas forman la base del pensamiento y razonamiento lógico que todo programador debe manejar de manera muy fluida. Por eso fundamental realizar mucha práctica sobre este tema .

Estructuras repetitivas - Ciclos

Los ciclos nos permiten repetir la ejecución de un código varias veces. Imaginemos que quisiéramos repetir la frase "Hola mundo" 5 veces. Podríamos hacerlo manualmente. Crea un archivo llamado loops.js y escribe el siguiente código:

```
console.log("Hola Mundo");
console.log("Hola Mundo");
console.log("Hola Mundo");
console.log("Hola Mundo");
console.log("Hola Mundo");
```

Ejecútalo en la consola del navegador y deberías ver la frase "Hola mundo" 5 veces en tu pantalla:

```
Hola mundo
Hola mundo
Hola mundo
Hola mundo
Hola mundo
```

Ahora imagina que quisiéramos repetir 1000 veces. Ya no sería tan divertido copiar todo ese número de líneas en el archivo. Podemos entonces utilizar un **ciclo**.

Ciclo While (Mientras)

Se crea utilizando la palabra clave **while** seguido de una **condición**, que va a definir el número de veces que se va a repetir ese ciclo. Reemplaza el contenido del archivo loops.js por el siguiente:

```
let i = 0;
```

```
while (i < 1000) {  
    console.log("Hola mundo");  
    i = i + 1;  
}
```

Ejecútalo y revisa que la frase "Hola mundo" aparezca 1000 veces. Como ejercicio modifícalo para que aparezca el valor de **i** antes de cada frase.

```
0 Hola mundo  
1 Hola mundo  
2 Hola mundo  
...  
345 Hola mundo  
...  
1000 Hola mundo
```

El ciclo while en JavaScript tiene la siguiente sintaxis:

```
while (<condicion>) {  
    // acá va el cuerpo del ciclo, el código que se va a repetir mientras la condición se cumpla  
}
```

La condición puede ser cualquier valor o expresión booleana. El cuerpo del ciclo se va a ejecutar mientras que la condición se cumpla. Por ejemplo, crea un archivo llamado infiniteLoop.js que contenga lo siguiente:

```
while (true) {  
    console.log("Hola Mundo");  
}
```

¿Qué consideran que va a ocurrir? El código anterior crea lo que en programación llamamos un **ciclo infinito**. Intenta evitarlos.

Si lo ejecutas en la consola del navegador, seguramente se quedará ejecutando de forma permanente consumiendo toda la memoria RAM que quede en tu computadora disponible

hasta que probablemente tengas que cerrar el navegador o la pestaña del mismo de manera forzada.

En el momento en el que la condición deja de cumplirse el ciclo se detiene y continúa con el resto del programa. Podemos crear un ciclo que nunca va a ejecutar el cuerpo del ciclo:

```
while (false) {  
    console.log("Hola mundo");  
}
```

Si ejecutas ese código no deberías ver ninguna frase "Hola mundo".

En vez de poner **true** o **false** puedes utilizar cualquier otra **condición** o **expresión booleana** como lo hicimos en el ciclo que muestra "Hola mundo" 1000 veces:

```
let i = 0;  
while (i < 1000) {  
    console.log("Hola mundo");  
    i++;  
}
```

Primero declaramos una variable **i** que inicia en 0. Cada vez que ingresa en el ciclo la vamos a incrementar en 1 hasta que lleguemos a 1000. En ese momento la condición va a dejar de ser verdadero y el ciclo se detendrá.

Ciclo Do While (Hacer Mientras)

Es muy similar al ciclo while. Este ciclo crea un bucle que ejecuta una sentencia especificada, hasta que la condición se evalúa como falsa. La condición se evalúa después de ejecutar el cuerpo del ciclo, dando como resultado que el cuerpo especificado **se ejecute al menos una vez**.

La siguiente sintaxis:

```
do{  
    // acá va el cuerpo del ciclo, el código que se va a repetir al menos una vez y mientras la  
    // condición se cumpla
```

```
}while (<condicion>)
```

La condición puede ser cualquier valor o expresión booleana. El cuerpo del ciclo se va a ejecutar al menos una vez y mientras que la condición se cumpla.

Por ejemplo

```
let i = 0;  
do {  
    console.log("Hola mundo");  
    i++;  
}  
while (i < 1000)
```

Primero declaramos una variable **i** que inicia en 0. Luego ingresamos al ciclo, imprimimos el mensaje e incrementamos el contador. Por último se evalúa la condición, en el caso de ser verdadera, se vuelve a ejecutar el ciclo.

Cada vez que ingresa en el ciclo la vamos a incrementar en 1 hasta que lleguemos a 1000. En ese momento la condición va a dejar de ser verdadero y el ciclo se detendrá.

Ciclo for

El while y do while es todo lo que necesitamos para hacer ciclos en JavaScript. Sin embargo, ese patrón que vimos en el ejemplo anterior en el que tenemos una **inicialización (let i = 0)**, una **condición (i < 850)** y un **incrementador o contador (i++)** es tan común, que JavaScript tiene un atajo para esto, el **for**.

El for tiene la siguiente sintaxis:

```
for (<inicialización>; <condición>; <incrementador>) {  
    // el cuerpo del ciclo, el código que se repite mientras que la condición sea verdadera  
}
```

El ejemplo anterior lo podemos reescribir de la siguiente forma:

```
for (let i=0; i < 1000; i++) {
```

```
console.log("Hola mundo");
}
```

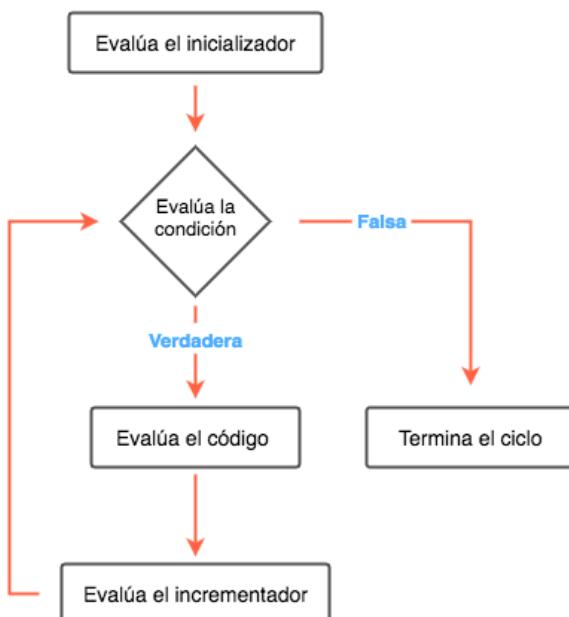
Son equivalentes, la única diferencia es que el **inicializador**, la **condición** y el **incrementador** están definidos en la misma línea, pero se ejecuta de la misma forma que el while:

La **inicialización** se ejecuta antes de evaluar la condición por primera vez.

La **condición** se ejecuta cada vez que se itera.

El **cuerpo** se ejecuta cada vez que la condición se cumple.

El **incrementador** o contador se ejecuta cada vez que el cuerpo se ejecuta, antes de volver a evaluar la condición.



Nota: Tanto el inicializador, la condición y el incrementador son opcionales. Si quisieras podrías hacer un ciclo infinito con un for de la siguiente forma:

```
for(;;) {
  // el cuerpo del ciclo también es opcional
}
```

```
}
```

Ejemplos con while y for

Imaginemos que queremos hacer un programa que imprima los números del 10 a 20 pero saltando cada otro número, es decir, que imprima 10, 12, 14, 16, 18 y 20.

El primer paso antes de escribir el ciclo es identificar las diferentes partes del ciclo: **la inicialización, la condición, el incrementador y el cuerpo**.

Para este ejemplo serían:

Inicializa una variable en 10.

La **condición** es que la variable sea menor o igual a 20.

Incrementa la variable en dos en cada iteración.

El **cuerpo** debe imprimir la variable.

Con esta información ya puedes implementar el ciclo con **while** o **for**, recuerda que son equivalentes. La solución utilizando un **while** sería:

```
let i = 10;    // el inicializador
while (i <= 20) { // la condición
  console.log(i); // el cuerpo
  i = i + 2;    // el incrementador
}
```

Podemos hacer lo mismo con un **for**:

```
for (let i=10; i <= 20; i = i + 2) {
  console.log(i);
}
```

Hagamos otro ejemplo. Imagina que queremos imprimir los números del 1 a al 100 pero de forma descendente, es decir, 100, 99, 98 ... 1. Empecemos identificando las partes del ciclo:

Inicializa una variable en 100.

La **condición** es que la variable sea mayor que 0.

El **incrementador** va a ser un **decrementador** en este caso, va a decrementar la variable en 1, en cada iteración.

El **cuerpo** debe imprimir la variable.

La solución con **while**:

```
let i = 100;
while (i > 0) {
    console.log(i);
    i--;
}
```

La solución con **for**:

```
for(let i=100; i > 0; i--){
    console.log(i);
}
```

Arreglos

Hasta ahora hemos trabajado con cadenas de texto, números y booleanos. En este apartado vamos a hablar de un nuevo tipo de datos: los **arreglos** (conocidos también como **arrays** o simplemente **listas**).

Un arreglo es una lista ordenada de elementos de cualquier tipo. Para crear tu primer arreglo abre la consola de navegador y escribe lo siguiente:

```
let array = [1, "Pedro", true, false, "Juan"]
```

La sintaxis de un arreglo es muy simple. Los elementos del arreglo se envuelven entre **corchetes** y se separan con **coma**. Veamos que el arreglo que creamos contiene números, cadenas de texto y booleanos.

Nota. En Javascript y en algunos otros lenguajes de programación, cada elemento del arreglo puede ser de cualquier tipo (incluso otros arreglos!).

Obtener elementos del arreglo

Para obtener la primera posición del arreglo que acabamos de crear utilizamos **array[0]**:

```
> array = [1, "Pedro", true, false, "Juan"]  
[1, "Pedro", true, false, "Juan"]  
> array[0]  
1
```

La sintaxis para obtener un elemento del arreglo es **[n]** donde **n** es la posición empezando en 0. Veamos de imprimir los demás elementos del arreglo:

```
> array[1]  
"Pedro"  
> array[2]  
true  
> array[3]  
false  
> array[4]  
"Juan"
```

Recorriendo un arreglo

En el ejemplo anterior pudimos imprimir cada una de las posiciones porque era un arreglo de pocos elementos. Sin embargo esto no siempre es práctico. Primero, el arreglo puede ser muy grande o puede que no sepamos el tamaño del arreglo. Crea un archivo llamado arrays.js y escribe el siguiente código:

```
let array = [1, "Pedro", true, false, "Juan"];
for (var i=0; i < array.length; i++) {
    console.log(array[i]);
}
```

Aquí simplemente usamos un ciclo for para mostrar cada uno de los elementos del arreglo.

Reemplazar un elemento

Es posible reemplazar el valor de cualquier elemento del arreglo. Por ejemplo:

```
let array = [1, "Pedro", true, false, "Juan"];
array[1] = "Germán"; // reemplazamos el elemento en la posición 1
// [1, "Germán", true, false, "Juan"]
```

En este ejemplo estamos reemplazando la posición 1 del arreglo (que realmente es la segunda porque recuerda que empieza en 0) con el valor "Germán". La línea más importante es la siguiente:

```
array[1] = "Germán";
```

Como ejercicio intenta reemplazar el último elemento ("Juan") por otro valor.

Agregar nuevos elementos

Es posible insertar nuevos elementos en un arreglo (puede estar vacío o tener elementos).

Por ejemplo:

```
let array = ["Pedro"];
array.push("Germán"); // ["Pedro", "Germán"]
array.push("Diana"); // ["Pedro", "Germán", "Diana"]
```

El método push te permite agregar un elemento al final de la lista. ¿Qué pasa si queremos agregar un elemento en otra posición? Para eso sirve el método **splice**:

```
var array = ["Pedro", "Germán", "Diana"];
array.splice(0, 0, "Juan") // ["Juan", "Pedro", "Germán", "Diana"]
```

El método **splice** se utiliza tanto para insertar como para eliminar elementos. Para insertar debes pasarle 3 o más argumentos. El primer argumento es la posición en la que quieres insertar el elemento. La segunda posición debe estar en 0. Los demás argumentos son los elementos que deseas insertar en el arreglo. Fíjate que en el ejemplo todos los elementos desde esa posición se mueven a la derecha.

Eliminar elementos

Para eliminar elementos de un arreglo utilizas el método **splice**. Por ejemplo:

```
let array = ["Pedro", "Germán", "Diana"];
array.splice(1, 1); // ["Pedro", "Diana"]
```

El método splice recibe uno o dos argumentos cuando se desea eliminar elementos: el índice del elemento que quieres eliminar y la cantidad de elementos a eliminar. Si omites el segundo argumento se eliminarán todos los elementos después del índice que hayas especificado en el primer argumento. Por ejemplo:

```
let array = ["Pedro", "Germán", "Diana"];
array.splice(0); // []
```

En este último ejemplo el arreglo queda vacío debido a que no se indicó el segundo parámetro del método splice.

Funciones

Frecuentemente vamos tener algunas líneas de código que necesitan ser ejecutadas varias veces y desde diferentes partes de nuestro programa. En vez de repetir el mismo código una y otra vez puedes crear una **función** (también conocidas como procedimientos o métodos) e invocarla cada vez que necesitemos ejecutar ese trozo de código. Por ejemplo si en un sistema de facturación web necesitamos tener que aplicar un descuento sobre un producto, y dicho descuento tiene que visualizarse en varias partes del sistema, entonces sería más conveniente crear una función que encapsula toda las líneas de código para calcular el descuento.

Entonces cómo creamos una función...

--crea un archivo llamado functions.js y escribe lo siguiente:

```
function hello() {  
    console.log("Hola Mundo");  
}
```

Para definir una función usamos la palabra reservada **function**, le damos un nombre (en este caso **hello**), abrimos y cerramos paréntesis `()`. Después abrimos corchetes `{}`, escribimos el **cuerpo** de la función (el código que queremos ejecutar cuando sea invocada), y por último cerramos los corchetes `}`.

Si ejecutamos este código usando la consola del navegador veremos que no aparece nada en la pantalla:

Una característica de las funciones es que no se ejecutan hasta que alguien las **invogue**.

Modifiquemos nuestro programa para invocarla:

```
function hello() {  
    console.log("Hola Mundo");  
}  
hello(); // acá la estamos invocamos
```

En la última línea la estamos invocando. Si lo ejecutamos ahora si debería aparecer "**Hola mundo**".

```
$ Hola mundo
```

Argumentos o Parámetros

Las funciones pueden recibir cero o más argumentos (o parámetros). Pensemos en los argumentos como variables que podemos utilizar dentro de la función. Utilizando argumentos podemos hacer por ejemplo una función reutilizable que salude a cualquier persona:

```
function hello(name) {  
    console.log("Hola " + name);  
}  
  
hello("Emilse");  
hello("Damian");
```

Si lo ejecutamos deberíamos ver lo siguiente:

```
$ Hola Emilse  
$ Hola Damian
```

Los argumentos se definen dentro de los **paréntesis** al declarar la función y se separan con **coma**.

Ejemplo de función con 2 argumentos:

```
function hello(firstName, lastName) {  
    console.log("Hola " + firstName + " " + lastName);  
}  
  
hello("Emilse", "Mendoza");  
hello("Damian", "Galetto");
```

Si ejecutamos el código anterior deberíamos ver lo siguiente:

```
$ Hola Emilse Mendoza  
$ Hola Damian Galetto
```

En la función anterior hemos dispuesto que reciba 2 parámetros **firstName** y **lastName**, cada uno de ellos separados por **(,)**. De esta manera si quisiéramos podríamos pasar más parámetros a la función. En el cuerpo de la función se concatena los dos parámetros para mostrar un mensaje a través de la consola.

Retornar un valor

Opcionalmente podemos retornar un valor desde la función utilizando la palabra clave **return**. Podemos modificar la función hello para que en vez de imprimir con console.log retorne una cadena de texto:

```
function hello(name) {  
    return "Hola " + name;  
}  
var name = hello("Emilse"); // podemos asignar el valor de retorno a una  
variable  
console.log(name);
```

```
// podemos llamar la función directamente en el parámetro de otra  
función.  
console.log(hello("Damian"));
```

En vez de hacer el console.log dentro de la función lo hacemos cuando la invocamos (de lo contrario no aparecería nada en pantalla).

En lo posible se recomienda retornar valores en vez de utilizar console.log dentro de las funciones. La razón es que retornar un valor hace la función más **reutilizable**. Ahora podemos utilizar esta función en otros contextos en donde no se utilice console.log para imprimir en la línea de comandos, como en una aplicación Web.

El **return es la última línea que se ejecuta de una función**, cualquier código que se encuentre después de esa línea será ignorado. Por ejemplo:

```
function hello(name) {  
    return "Hola " + name;  
    console.log("Esto nunca se va a imprimir");  
}  
console.log(hello("Emilse"));
```

Si ejecutas este código deberás ver lo siguiente:

```
$ Hola Emilse
```

La última línea de la función nunca va a ser ejecutada porque la función siempre retorna antes de llegar a ella.

Estructura de una función

Recapitulemos lo que hemos visto hasta ahora. La sintaxis de una función es la siguiente:

```
function <name>([arg1], [arg2], ...) {  
    // cuerpo de la función  
    return <valor de retorno>;  
}
```

Lo que debes tener en cuenta:

- La función se crea con la palabra clave **function**.
- El nombre de la función tiene las mismas reglas de nombramiento que las variables: debe comenzar con \$, _ o una **letra**, y después puede contener **letras, dígitos, _ y \$**.
- La función puede tener cero o más argumentos dentro de los paréntesis que van después del nombre.
- Pensemos en los argumentos como variables que puedes utilizar en la función.
- Los valores de esos argumentos se definen cuando invocan la función.
- Cada argumento debe tener un nombre de una variable válido. Recordemos que el nombre de una variable debe comenzar con \$, _ o una **letra**, y después puede contener **letras, dígitos, _ y \$**.
- Podemos retornar un valor desde la función utilizando la palabra clave **return**.
- El valor de retorno debe ser un tipo válido de JavaScript: un **número**, una **cadena de texto**, un **booleano**, un **arreglo**, etc.
- Puedes almacenar el valor de retorno de una función en una variable o puedes invocar la función como parámetro de otra función.

Cajas negras

En muchas ocasiones es bueno asociar a las funciones como cajas negras que reciben unos parámetros de entrada y genera un valor de salida (el valor de retorno).

Ejemplo

Vamos a hacer una función que calcule el índice de masa corporal (IMC). El IMC es una medida que relaciona el peso de una persona con su altura. La fórmula para calcular el IMC es peso dividido altura al cuadrado:

$$\text{IMC} = \text{peso} / (\text{altura}^2)$$

Traduzcamos eso a código JavaScript. Crea un archivo llamado imc.js y escribe lo siguiente:

```
function imc(weight, height) {  
    return weight / height**2  
}  
console.log("Tu IMC es: " + bmi(80, 1.8));
```

Si ejecutamos el archivo debería mostrar algo así:

```
$ Tu IMC es: 24.6913580
```

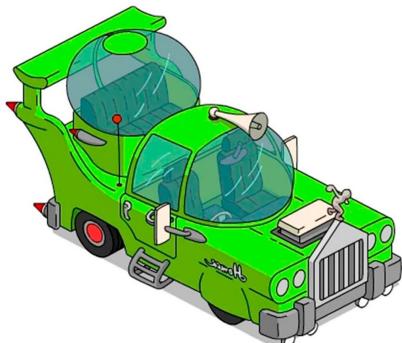
Objetos

JavaScript está diseñado sobre un paradigma simple basado en objetos. Un objeto es una colección de propiedades, y una propiedad es una asociación entre un nombre (o clave) y un valor. Un valor de una propiedad puede ser una función, en cuyo caso la propiedad es conocida.

Los objetos en JavaScript, como en tantos otros lenguajes de programación, pueden ser comparados con objetos de la vida real. El concepto de Objetos en JavaScript puede entenderse con objetos tangibles de la vida real.

En JavaScript, un objeto es una entidad independiente con propiedades y tipos. Compáralo con un auto, por ejemplo. Un auto es un objeto con propiedades. Una auto tiene un color, una marca, un modelo, año de fabricación, potencia, etc. De la misma manera, los objetos de JavaScript pueden tener propiedades, las cuales definen sus características.

Veamos un primer ejemplo:



Propiedades de un auto:

- color
- marca
- modelo
- año
- potencia

En código Javascript :

```
var auto = {  
    color: "verde",  
    marca: "fiat",  
    modelo: "bravo",  
    año: 2008,  
    potencia: "130bhp"  
}
```

En este ejemplo estamos creando un objeto y lo estamos almacenando en la variable **auto**.

Un objeto se define utilizando corchetes **{}**. Las propiedades se separan con coma (,) y las llaves y valores se separan con dos puntos (:).

En este objeto estamos almacenando la información de un auto, pero en un objeto podemos almacenar cualquier tipo de información que requiera esa asociación llave-valor.

El valor de una propiedad puede ser cualquier tipo de datos en JavaScript: números, cadenas de texto, booleanos, arreglos e incluso funciones y otros objetos.

Obtener valores de un objeto

Para obtener el color del **auto** en el objeto que definimos previamente lo haríamos utilizando **auto.color**.

Por ejemplo:

```
console.log(auto.color);
```

Para obtener el valor de una llave en un objeto utilizamos la notación punto (.), es decir, el **nombre de la variable**, seguido de **punto**, seguido del **nombre de la llave**:

```
console.log(auto.color); // imprime rojo
console.log(auto.marca); // imprime fiat
console.log(auto.modelo); // imprime bravo
console.log(auto.año); // imprime 2008
console.log(auto.potencia); // imprime 130bhp
```

Existe otra forma equivalente de obtener el valor de una llave utilizando corchetes cuadrados ([]):

```
auto["color"]
```

Esta notación es útil para obtener los valores de forma dinámica. Por ejemplo:

```
var llave = "color";
console.log(auto[llave]);
```

Primero definimos una variable **llave**, a la que le asignamos el valor "color" y utilizamos esa variable para obtener el valor. Esto va a ser útil más adelante cuando estemos recorriendo las propiedades de un objeto.

Como ejercicio intenta cambiar el valor de la variable por cualquier otra llave del objeto y verifica el resultado. ¿Qué pasa si utilizamos una llave que no existe? ¡Inténtalo!

Agregar nuevas propiedades al objeto

Es posible agregar más propiedades a un objeto después de que ha sido creado. Por ejemplo, podríamos agregar una propiedad con llave **precio** y valor 70000 de la siguiente forma:

```
auto.precio = 70000;
```

Modificar propiedades del objeto

Es también posible modificar los valores de las propiedades de un objeto. Por ejemplo, si queremos cambiar el valor de la llave precio lo podemos hacer de la siguiente forma:

```
auto.precio = 65000;
```

Eliminar propiedades de un objeto

Para eliminar una propiedad de un objeto utiliza el operador **delete**:

```
delete auto.precio;
```

Recorrer las propiedades de un objeto

Existen varias formas de recorrer las propiedades de un objeto en JavaScript. Veamos las dos principales:

```
for (var llave in auto) {
  if (persona.hasOwnProperty(llave)) {
    console.log(auto[llave])
  }
}
```

La razón por la que tenemos que agregar la condición es que los objetos pueden heredar propiedades de otros objetos, pero esa explicación se verá más adelante en este manual. Por ahora deberemos asegurar de agregar el condicional si vamos a utilizar esa forma de iterar.

La otra forma de recorrer las propiedades de un objeto es utilizando **Object.keys**:

```
var llaves = Object.keys(auto);
for (var i=0; i < llaves.length; i++) {
  var llave = llaves[i];
  console.log(auto[llave]);
}
```

Object.keys retorna un arreglo con las llaves del objeto que almacenamos en la variable **llaves**. Después iteramos por todas las llaves y utilizamos cada llave para obtener, de forma dinámica, el valor de esa llave en el objeto.

Ejemplo de arreglos y objetos

Es posible mezclar arreglos y objetos para crear estructuras complejas. Crea un archivo llamado **products.js** y transcribe lo siguiente:

```
var products = [
  { id: 1, name: "Leche", price: 120, categories: ["familiar", "comida"] }
```

```
},
  { id: 2, name: "Arroz", price: 80, categories: ["familiar", "comida"] },
  { id: 3, name: "Lavadora", price: 7800, categories:
  ["electrodomésticos"] }
];
```

En este ejemplo hemos creado un arreglo de objetos. Cada objeto representa un producto y una de sus llaves (**categories**) contiene a su vez un arreglo. Modifiquemos el programa para imprimir los productos en la consola:

```
var products = [
  { id: 1, name: "Leche", price: 120, categories: ["familiar", "comida"] },
  { id: 2, name: "Arroz", price: 80, categories: ["familiar", "comida"] },
  { id: 3, name: "Lavadora", price: 7800, categories:
  ["electrodomésticos"] }
];

for (var i=0; i < products.length; i++) {
  var product = products[i];
  console.log(product.name);
  console.log("  Id: " + product.id);
  console.log("  Precio: " + product.price);
  console.log("  Categorías: " + product.categories.join(", "));
}
```

Lo primero que estamos haciendo es iterando por el arreglo de **productos**. Por cada uno de los productos (recuerda que cada producto es un objeto) vamos a mostrar el nombre (la llave **name**), después el identificador (la llave **id**), el precio (la llave **price**) y las categorías (la llave **categories**). Como las categorías están en un arreglo debemos utilizar el método **join** para convertirlas en una cadena.

Formas de crear objetos

Hasta el momento vimos como crear objetos denominados **literales**.

Con notación literal

```
var auto = {  
    color: rojo,  
    marca: "fiat",  
    modelo: "bravo",  
    año: 2008  
}
```

Pero existen otras formas de crear objetos en Javascript que los veremos a continuación.

Con función constructora

La notación literal nos permite crear un único objeto mientras que la función constructora es una plantilla con la cual podemos crear varios objetos. Normalmente, para diferenciarla de otras funciones se la suele nombrar con mayúscula.

```
//Función constructora de objetos  
function Auto(color, marca, modelo, año){  
    this.color = color;  
    this.marca = marca;  
    this.modelo = modelo;  
    this.año = año;  
}  
  
var auto = new Auto("rojo", "fiat", "bravo", 2008 );  
console.log(auto.color);
```

El constructor Object()

El objeto global que tiene Javascript Object nos permite crear objetos vacíos genéricos. Luego podemos añadirle las propiedades que nosotros deseamos.

```
var auto = new Object();  
auto.color = "rojo";  
auto.marca = "fiat";  
auto["modelo"] = "bravo",  
auto.año = 2008;
```

Object.create

Además de la notación literal y de la función constructora, Javascript también permite crear objetos con el método `Object.create()`. Esto es especialmente útil cuando sabemos de antemano que no vamos a crear demasiadas instancias de un objeto. Javascript tiene un método en el objeto global **Object** denominado **create** que nos permite crear un nuevo objeto basado en uno pre-existente. Veamos un ejemplo:

Creamos un nuevo objeto **auto2** usamos el objeto **auto** creado en el apartado anterior

```
var auto2 = Object.create(auto);
console.log(auto2.marca) //imprime fiat
```

Vemos que **auto2** tiene acceso a todas las propiedades y métodos que se hayan definido en **auto**. Además crear nuevas propiedades si quisieramos.

```
auto2.precio = 45000;
```

Comparando Objetos

Los objetos son de tipo referencia en JavaScript. Dos objetos con las mismas propiedades y métodos nunca son iguales. Sólo comparando la misma referencia al objeto consigo mismo dará como resultado true.

```
// variable de referencia del objeto persona1
var persona1 = {nombre: "Juan"};

// variable de referencia del objeto persona2
var persona2 = {nombre: "Juan"};

persona1 == persona2 // retorna false
persona1 === persona2 // retorna true
```

Veamos otro ejemplo:

```
// variable de referencia del objeto persona1
var persona1 = {nombre: "Juliana"};

// variable de referencia del objeto persona2
var persona2 = persona1; // asignamos la referencia del objeto persona a
la variable de referencia del objeto persona2
```

```
// aquí persona1 y persona2 apuntan al mismo objeto Llamado persona1
persona1 == persona2 // retorna true

// aquí persona1 y persona2 apuntan al mismo objeto Llamado persona1
persona1 === persona2 // retorna true
```

Entonces dos objetos son iguales si solo si su referencia es la misma.

Nota: El operador "===" se utiliza para comprobar el valor así como el tipo

Objetos globales

Son los objetos que tenemos disponibles en el ámbito global (objetos primitivos, es decir, que viene con el lenguaje).

Los podemos dividir en 3 grupos

- **Objetos contenedores de datos:** Object, Array, Function, Boolean, Number
- **Objetos de utilidades:** Math, Date, RegExp
- **Objetos de errores:** Error

Object

Es el padre de todos los objetos Javascript, es decir, cualquier objeto hereda de él
Para crear un objeto vacío podemos usar:

- La notación literal : var o = {}
- La función constructora Object(): var o = new Object();

Un objeto contiene las siguientes propiedades y métodos:

- La propiedad **o.constructor** con la función constructora
- El método **o.toString()** que devuelve el objeto en formato texto
- El método **o.valueOf()** que devuelve el valor del objeto (normalmente o)

Ejemplo:

```
var o = new Object();
o.toString()
"[object Object]"
o.valueOf() === o
true
```

Array

Para crear arrays podemos usar:

- La notación literal : `var a = []`
- La función constructora `Array()`: `var o = new Array();`

Podemos pasarle parámetros al constructor `Array()`

- Varios parámetros: Serán asignados como elementos al array
- Un numero: Se considerará el tamaño del array

Ejemplo:

```
> var a = new Array(1,2,3, 'luna');
> a;
> [1, 2, 3, "four"]
> var a2 = new Array(5);
> a2;
> [undefined, undefined, undefined, undefined, undefined]
```

Como los arrays son objetos tenemos disponibles los métodos y propiedades del padre `Object()`:

```
> typeof a;
> "object"
> a.toString();
> "1,2,3,luna"
> a.valueOf()
> [1, 2, 3, "luna"]
> a.constructor
> Array()
```

Los arrays disponen de la propiedad `length`

- Nos devuelve el tamaño del array (numero de elementos)
- Podemos modificarlo y cambiar el tamaño del array

```
> a[0]=1;
> a.length
> 1
> a.length = 5
```

```
> a
> [1, undefined, undefined, undefined, undefined]
> a.length = 2;
> 2
> a
> [1, undefined]
```

Los arrays disponen de unos cuantos métodos interesantes:

- push()
- pop()
- sort()
- join()
- slice()
- splice()

Veamos rápidamente cada uno de ellos

Push

push() inserta elementos al final del array **a.push('new')** es lo mismo que **a[a.length] = 'new'**
push() devuelve el tamaño del array modificado.

```
> var sports = ['soccer', 'baseball'];
> sports
> ["soccer", "baseball"]
> sports.length
> 2
> sports.push('football', 'swimming');
> 4
> sports
> ["soccer", "baseball", "football", "swimming"]
```

Pop

pop() elimina el último elemento, **a.pop()** es lo mismo que **a.length-=1**; pop() devuelve el elemento eliminado.

```
> var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];
> myFish
> ["angel", "clown", "mandarin", "sturgeon"]
> myFish.pop();
> "sturgeon"
> myFish
> ["angel", "clown", "mandarin"]
```

Sort

Ordena el array y devuelve el array modificado

```
> var fruit = ['apples', 'bananas', 'Cherries'];
> fruit
> ["apples", "bananas", "Cherries"]
> fruit.sort();
> ["Cherries", "apples", "bananas"]

> var scores = [1, 2, 10, 21];
> scores
> [1,2,10,21]
> scores.sort()
> [1, 10, 2, 21]
```

Nota: El método **sort** puede ordenar alfabéticamente o numéricamente, en forma ascendente o descendente. Por defecto sort ordena los valores como **strings** y trata los valores como caracteres alfabéticos y en forma **descendente**. Notamos que esto trabaja bien para strings (Por ejemplo **Apple** está antes que **Banana**). Sin embargo, si los valores a ordenar son números, y son tratados como strings, “25” es mayor a “100” porque 2 es mayor que 1 si son tratados como strings. Entonces esto nos puede generar un resultado incorrecto para ordenar números. Para solucionar este problema se usa una función de comparación

Usando función de comparación

Ordenamiento de números en forma ascendente:

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){
    return a-b
```

```
});
```

El resultado sería:

```
> [1,5,10,25,40,100];
```

Ordenamiento de números en forma descendente:

```
var points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){
    return b-a
});
```

El resultado sería:

```
> [100,40,25,10,5,1];
```

En los ejemplos anteriores vemos que el método sort recibe una función como parámetro. Dicha función se conocen como funciones anónimas y en este caso recibe 2 parámetros. Sort internamente usa algún [método de ordenamiento](#) conocidos en la teoría de fundamentos de programación. Sobre funciones anónimas veremos más adelante.

Join

join() devuelve una cadena (string) con los valores de los elementos del array

```
> var a = ['Wind', 'Rain', 'Fire'];
> a.join();
> "Wind,Rain,Fire"
> a.join(" - ");
> "Wind - Rain - Fire"
> typeof ( a.join(" - ") )
> "string"
```

Slice

slice() devuelve un trozo del array sin modificar el original

```
> var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];
> var citrus = fruits.slice(1, 3);
> fruits
> ["Banana", "Orange", "Lemon", "Apple", "Mango"]
> citrus
> ["Orange", "Lemon"]
```

Splice

splice() quita un trozo del array, lo devuelve y opcionalmente rellena el hueco con nuevos elementos

```
var myFish = ['angel', 'clown', 'mandarin', 'surgeon']; 2
myFish
["angel", "clown", "mandarin", "surgeon"]
var removed = myFish.splice(2, 1);
myFish
["angel", "clown", "surgeon"]
removed
["mandarin"]
var removed = myFish.splice(2, 0, 'drum');
myFish
["angel", "clown", "drum", "surgeon"]
removed
[]
```

Function

Las funciones son objetos

Podemos crear funciones con la función constructora Function() (aunque este método no se recomienda ya que internamente hace un [eval\(\)](#))

```
function sum(a, b) {
  return a + b;
}
> sum(1, 2)
> 3

var sum = function(a, b) {
  return a + b;
```

```
};

> sum(1, 2)
> 3

> var sum = new Function('a', 'b', 'return a + b;');
> sum(1, 2)
> 3
```

Las funciones disponen de las siguientes propiedades:

- La propiedad **constructor** que contiene una referencia a la función constructora `Function()`
- La propiedad **length** que contiene el numero de parametros que acepta la función
- La propiedad **caller** (no standard) que contiene una referencia a la función que llamó a esta función
- La propiedad **prototype** que contiene un objeto.
 - Sólo es util cuando utilizamos una función como constructora.
 - Todos los objetos creados con una función constructora mantienen una referencia a su propiedad `prototype` y pueden usar sus propiedades como si fueran propias.

```
> function myfunc(a){ return a; }
> myfunc.constructor
> Function()
```

```
> function myfunc(a, b, c){ return true; }
> myfunc.length
> 3
```

```
> function A(){ return A.caller; }
> function B(){ return A(); }
> B()
B() //imprime B porque es la referencia desde donde se llama a A()
```

Las funciones disponen de los siguientes métodos:

- El método **toString()** que devuelve el código fuente de la función
- Los métodos **call()** y **apply()** que ejecutan métodos de otros objetos especificando el contexto (específicamente un **this** diferente). Estos dos métodos hacen lo mismo pero el formato en que reciben los argumentos es diferente

```
> function myfunc(a, b, c) {return a + b + c;}
> myfunc.toString()
"function myfunc(a, b, c) {
  return a+b+c;
}"

var some_obj = {
  name : 'Ninja',
  say: function(who){
    return 'Haya dude ' + who + ' I am a ' + this.name;
}
}
> some_obj.say('Dude');
"Haya Dude, I am a Ninja"
> my_obj = {name: 'Scripting guru'};
> some_obj.say.call(my_obj, 'Dude');
"Haya Dude, I am a Scripting guru"
> some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
> some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

Boolean

El objeto Boolean es un contenedor para un valor de tipo booleano Podemos crear objetos Boolean con la función constructora Boolean()

```
> var b = new Boolean();
> typeof b
"object"
> typeof b.valueOf()
"boolean"
> b.valueOf()
false
```

La función Boolean usada como función normal (sin new) nos devuelve el valor pasado como parametro convertido a booleano.

```
> Boolean("test")
true
> Boolean("")
false
> Boolean({})
```

```
true
```

Number

La función Number() puede ser usada:

- Cómo una función normal para convertir valores a número
- Cómo una función constructora (con new) para crear objetos

Los objetos número disponen de los métodos:

- toFixed()
- toPrecision()
- toExponential()

```
> var n = new Number(123.456)
> n.toFixed(1)
"123.5"
> (12345).toExponential()
"1.2345e+4"
```

El método **toString()** de un objeto numero nos permite transformar un numero a una base determinada.

```
> var n = new Number(255);
> n.toString();
"255"
> n.toString(10); //base decimal
"255"
> n.toString(16); //base hexadecimal
"ff"
> (3).toString(2); //base binaria
"11"
> (3).toString(10); //base decimal
"3"
```

String

Podemos crear objetos String con la función constructora String()
Un objeto String **NO** es un dato de tipo primitivo string (valueOf())

```
> var primitive = 'Hello';
> typeof primitive;
"string"
> var obj = new String('world');
> typeof obj;
"object"
> Boolean("")
false
> Boolean(new String(""))
true
```

Un objeto string es parecido a un array de caracteres:

- Cada carácter tiene una posición indexada
- Tiene disponible la propiedad length

```
> var obj = new String('world');
> obj[0]
"w"
> obj[4]
"d"
> obj.length
5
```

Aunque los métodos pertenezcan al objeto String, podemos utilizarlos también directamente en datos de tipo primitivo string (se crea el objeto internamente).

```
> "potato".length
6
> "tomato"[0]
"t"
> "potato"[ "potato".length - 1]
"o"
```

Los objetos string disponen de los siguientes métodos:

- **toUpperCase()** devuelve el string convertido a mayúsculas
- **toLowerCase()** devuelve el string convertido a minusculas
- **charAt()** devuelve el carácter encontrado en la posición indicada
- **indexOf()** busca una cadena de texto en el string y devuelve la posición donde la encuentra
 - Si no encuentra nada devuelve -1
- **lastIndexOf()** empieza la búsqueda desde el final de la cadena

-
- Si no encuentra nada devuelve -1

Por lo tanto la manera de correcta de chequear si existe una cadena de texto en otra es:

```
if (s.toLowerCase().indexOf('couch') !== -1) {...}
```

Otros métodos útiles para objetos String:

- **slice()** devuelve un trozo de la cadena de texto
- **split()** transforma el string en un array utilizando un string como separador
- **concat()** une strings

Veamos un ejemplo de todos estos métodos:

```
> var s = new String("Couch potato");
> s.toUpperCase()
"COUCH POTATO"
> s.toLowerCase()
"couch potato"
> s.charAt(0);
"C"
> s.indexOf('o')
1
> s.lastIndexOf('o')
11
> s.slice(1,5)
"ouch"
> s.split(" ")
["Couch", "potato"]
> s.concat("es")
"Couch potatoes"
```

Math

Math es un objeto con propiedades y métodos para usos matemáticos. Math NO es un constructor de otros objetos.

Algunos métodos interesantes de Math son:

- **random()** genera números aleatorios entre 0 y 1
- **round(), floor() y ceil()** para redondear números
- **min() y max()** devuelven el mínimo y el máximo de una serie de números pasados como parametros
- **pow() y sqrt()** devuelve la potencia y la raíz cuadrada respectivamente

Ejemplo generar número aleatorio hasta cierto número:

```
var maxNumber = 20;  
Math.round(Math.random() * maxNumber);
```

Ejemplo generar número aleatorio entre 2 números:

```
var max = 1;  
var min = 100;  
Math.round(Math.random() * (max - min) + min);
```

Date

`Date()` es una función constructora que crea objetos de tipo fecha.

Podemos crear objetos Date pasándole:

- **Nada** (tomará por defecto la fecha actual)
- **Una fecha** en formato texto
- **Valores** separados que representan: Año, Mes(0-11), Dia(1-31), Hora(0-23), Minutes(0-59),
- Segundos(0-59) y Milisegundos (0-999)
- Un valor **timestamp**

Veamos algunos ejemplos:

```
> new Date()  
Fri Aug 23 2019 12:08:35 GMT-0300 (hora estándar de Argentina)  
> new Date('2019 08 23')  
Fri Aug 23 2019 00:00:00 GMT-0300 (hora estándar de Argentina)  
> new Date(2019,7,23,12,11,30,200)  
Fri Aug 23 2019 12:11:30 GMT-0300 (hora estándar de Argentina)  
> new Date(1566573276000)  
Fri Aug 23 2019 12:14:36 GMT-0300 (hora estándar de Argentina)
```

Para ver más sobre fechas en formato timestamp ver: [Marca Temporal](#) , [Convertidor de fechas](#)

Algunos métodos para trabajar con objetos Date son:

- **setMonth()** y **getMonth()** escriben y leen el mes en un objeto date respectivamente (lo mismo disponemos métodos para year, day, hours, minutes, etc...)

-
- **parse()** dado un string, devuelve su timestamp
 - **UTC()** produce un timestamp dados un año, mes, dia, etc..
 - **toDateString()** devuelve el contenido de un objeto date en formato americano

Veamos algunos ejemplos:

```
> var d = new Date();
> d.toString();
"Fri Aug 23 2019 12:20:29 GMT-0300 (hora estándar de Argentina)"
> d.setMonth(2);
1553354429004
> d.toString();
"Sat Mar 23 2019 12:22:30 GMT-0300 (hora estándar de Argentina)"
> d.getMonth();
2

> Date.parse('Aug 23, 2019')
1566529200000
> Date.parse(2019,7,23)
1566529200000
```

```
> var d = new Date(2019,7,23)
> d.getDay();
5
> d.toDateString();
"Fri Aug 23 2019"
```

Cuando trabajamos con fechas tenemos que tener en cuenta que según la ubicación geográfica del usuario que accede a nuestro sistema, puede que usen un formato u otro para visualizar las fechas. En América por ejemplo se usa mucho el formato DD/MM/YYYY (día, mes , año) para presentar las fechas.

Veamos un ejemplo de cómo aplicar el formato DD/MM/YYYY a una fecha :

```
> var d = new Date()
> var dateFormatted = d.getDate() + "/" + (d.getMonth() + 1) + "/" +
d.getFullYear()
> console.log(dateFormatted)
"23/8/2019"
```

Otras funciones útiles para visualizar una fecha:

```
console.log("1") + new Date().toDateString());
console.log("2") + new Date().toISOString();
console.log("3") + new Date(). toJSON();
console.log("4") + new Date().toLocaleDateString();
console.log("5") + new Date().toLocaleString();
console.log("6") + new Date().toLocaleTimeString();
console.log("7") + new Date().toString();
console.log("8") + new Date().toISOString().slice(0,10));
```

La salida del ejemplo anterior sería:

```
1) Fri Aug 23 2019
2) 2019-08-23T15:45:07.649Z
3) 2019-08-23T15:45:07.650Z
4) 23/8/2019
5) 23/8/2019 12:45:07
6) 12:45:07
7) Fri Aug 23 2019 12:45:07 GMT-0300 (hora estándar de Argentina)
8) 2019-08-23
```

Más adelante en este curso usaremos una librería de javascript muy conocida para trabajar con fechas: [moment.js](#)

Programación Orientada a Objetos (POO)

La programación orientada a objetos (**POO**) es un paradigma de programación que utiliza la abstracción para crear modelos basados en el mundo real. Hoy en día, muchos lenguajes de programación (como Java, JavaScript, C#, C++, Python, PHP, Ruby y Swift) soportan programación orientada a objetos (POO).

La POO puede considerarse como el diseño de software a través de un conjunto de objetos que cooperan, que interactúan y comparten información; a diferencia de un punto de vista tradicional en el que un programa puede considerarse como un conjunto de funciones, o simplemente como una lista de instrucciones ejecutadas por la computadora. En este paradigma, cada objeto es capaz de recibir mensajes, procesar datos y enviar mensajes a otros objetos. Cada objeto puede verse como una pequeña máquina independiente con un papel o responsabilidad definida.

La POO pretende promover una mayor flexibilidad y facilidad de mantenimiento en la programación y es muy popular en la ingeniería de software a gran escala. Gracias a su fuerte énfasis en la modularidad, el código orientado a objetos está concebido para ser más fácil de desarrollar y más fácil de entender posteriormente.

Elementos de la POO

- Clase
 - Constructor
 - Propiedades
 - Métodos
 - Instancias
- Herencia
- Encapsulación
- Abstracción
- Polimorfismo

Clase

JavaScript es un lenguaje basado en prototipos que no contiene ninguna declaración de clase, como se encuentra, por ejemplo, en C++ o Java. Esto es a veces confuso para los programadores acostumbrados a los lenguajes con una declaración de clase. En su lugar, JavaScript utiliza funciones como clases. Definir una clase es tan fácil como definir una función.

Una clase es un contenedor que define las propiedades y métodos que van a tener los objetos creados a partir de este contenedor. Visto de otra forma una clase sería como una plantilla que usamos como modelo para crear objetos a partir de esta.

Para definir una clase usamos el concepto de función constructora que vimos anteriormente.

Ejemplo:

```
function Persona(){  
}
```

Aquí estamos definiendo una clase con la sintaxis de una función constructora. Esta clase se podría definir como una plantilla vacía ya que no tiene ni propiedades ni métodos.

Veamos otro ejemplo:

```
function Persona(nombre, apellido, edad){  
    this.nombre = nombre;  
    this.apellido = apellido;  
    this.edad = edad;
```

```
this.comer = function(){
    alert('Comiendo...');
}
}
```

En el ejemplo anterior, la clase **Persona** contiene las propiedades nombre, apellido y edad y un método **comer()**. Esto significa que cada objeto que vayamos a crear de esta clase, tendremos que pasarle esos parámetros.

Definiendo clases usando ES6

```
class Persona {  
}
```

Para definir una clase usamos la palabra clave **class** seguido del nombre de la clase que queremos crear. Luego entre {} vamos a colocar todas propiedades y métodos que pueda tener la clase.

```
// Definimos la clase
class Persona {
    constructor(nombre,apellido,edad) {
        this.nombre= nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    //Definimos un método
    comer(){
        alert('comiendo');
    }
}
```

En el ejemplo anterior usamos la especificación ES6 para definir una clase usando la palabra reservada **class** seguido nombre de la clase.

Objetos e instancias

Los objetos son entidades lógicas que se crean a partir de una clase.

Ejemplo:

```
var persona1 = new Persona("Juan", "Mendoza", 20);
var persona2 = new Persona("Carlos", "Gardel", 24);

console.log(persona1.nombre + ' - ' + persona1.apellido);
console.log(persona2.nombre + ' - ' + persona2.apellido);
```

Si ejecutamos en el navegador vamos a ver lo siguiente:

```
$ Juan Mendoza
$ Carlos Gardel
```

En este ejemplo estamos creando dos objetos de la clase **Persona**. Para crear cada objeto usamos la palabra reservada de Javascript **new**, seguido del nombre de la clase y los parámetros que espera la función constructora. A este proceso también se lo conoce como creación de **instancia** de una clase.

Constructor

El constructor es llamado en el momento de la creación de la instancia (el momento en que se crea la instancia del objeto). El constructor **es un método de la clase**. En JavaScript, la función sirve como el constructor del objeto, por lo tanto, no hay necesidad de definir explícitamente un método constructor. Cada acción declarada en la clase es ejecutada en el momento de la creación de la instancia.

El constructor se usa para establecer las propiedades del objeto o para llamar a los métodos para preparar el objeto para su uso.

En el siguiente ejemplo, el constructor de la clase Persona muestra un alerta que dice (Una instancia de la clase Persona) cuando se crea la instancia de la clase Persona.

```
function Persona() {
  alert('Una instancia de la clase Persona');
}

var persona1 = new Persona();
var persona2 = new Persona();
```

Veamos ahora cómo se define un constructor usando ES6:

```
class Persona {  
    constructor(nombre, apellido){  
        this.nombre = nombre;  
        this.apellido = apellido;  
    }  
}
```

Vemos en el ejemplo anterior que se declara un método denominado **constructor** que representa el constructor de la clase. Dicho método recibe dos parámetros **nombre** y **apellido** que servirán para inicializar las propiedades de los objetos que se creen de la clase **Persona**.

Algunas consideraciones sobre los constructores en ES6:

- Sólo puede haber un método especial con el nombre de "**constructor**" en una clase. Un error de sintaxis será lanzada, si la clase contiene más de una ocurrencia de un método constructor.
-
- Un constructor puede utilizar la palabra clave `super` para llamar al constructor de una clase padre.(Esto lo veremos más adelante)
-
- Si no especifica un método constructor, se utiliza un constructor predeterminado.
 - Para una clase base:
`constructor() {}`
 - Para una clase derivada:
`constructor(...args) {
 super(...args);
}`

Propiedades

Las propiedades son variables contenidas en la clase, cada instancia del objeto tiene dichas propiedades.

Para trabajar con propiedades dentro de la clase se utiliza la palabra reservada **this**, que se refiere al **objeto actual**. El acceso (lectura o escritura) a una propiedad desde fuera de la clase se hace con la sintaxis: **NombreDeLaInstancia.Propiedad** (Desde dentro de la clase la sintaxis es **this.Propiedad**). **this** se utiliza para obtener o establecer el valor de la propiedad).

En el siguiente ejemplo definimos la propiedad **nombre** de la clase **Persona** y la definimos en la creación de la instancia.

```
function Persona(nombre) {  
    this.nombre = nombre; //definición de la propiedad nombre  
}  
  
var persona1 = new Persona("Susan");  
var persona2 = new Persona("Michael");  
  
alert ('persona1 es ' + persona1.nombre); // muestra "persona1 es Susan"  
alert ('persona2 es ' + persona2.nombre); // muestra "persona2 es Michael"
```

Métodos

Los métodos siguen la misma lógica que las propiedades, la diferencia es que son funciones y se definen como funciones. Llamar a un método es similar a acceder a una propiedad, pero se agrega () al final del nombre del método, posiblemente con argumentos.

En el siguiente ejemplo se define y utiliza el método **saludar()** para la clase **Persona**.

```
function Persona(nombre) {  
    this.nombre = nombre;  
}  
  
Persona.prototype.saludar = function() {  
    alert ('Hola, Soy ' + this.nombre);  
};  
  
var persona1 = new Persona("Jorge");  
var persona2 = new Persona("Emilse");  
  
// Llamadas al método saludar de la clase Persona.  
persona1.saludar(); // muestra "Hola, Soy Jorge"  
persona2.saludar(); // muestra "Hola, Soy Emilse"
```

Definiendo métodos en ES6

```
class Persona {  
    constructor(nombre){  
        this.nombre = nombre;
```

```
}

saludar(){
    alert('Hola, Soy ' + this.nombre);
}
}

var persona1 = new Persona("Jorge");

// Llamadas al método saludar de la clase Persona.
persona1.saludar(); // muestra "Hola, Soy Jorge"
```

Settes y Getters

Los getters y los setters son construcciones habituales de los objetos que permiten acceder a valores o propiedades, sin revelar la forma de implementación de las clases. En pocas palabras, permiten encapsular los objetos y evitar mantenimiento de las aplicaciones cuando la forma de implementar esos objetos cambia.

En la práctica son simplemente métodos que te permite acceder a datos de los objetos, para leerlos o asignar nuevos valores. El setter lo que hace es asignar un valor y el getter se encarga de recibir un valor.

Propiedades computadas de objetos JavaScript

Get y set son prácticos para el acceso a propiedades de los objetos que resultan de computar otras propiedades que ya existen. Como ejemplo, vamos a crear el objeto **Persona**. Le añadimos las propiedades **nombre** y **apellido**. El resultado de nombre y apellidos sería su nombre completo, pero sin ser una nueva propiedad.

Ejemplo usando la sintaxis de objeto literal

```
var persona = {
    nombre: 'Pancho',
    apellido: 'Villa',
    get nombreCompleto() {
        return this.nombre + ' ' + this.apellido;
    }
}
```

Ejemplo usando class de ES6:

```
class Persona {
    constructor(nombre, apellido){
```

```
        this.nombre = nombre;
        this.apellido = apellido;

    }
    get nombreCompleto(){
        return this.nombre + ' ' + this.apellido;
    }
}
```

Get nos ha servido para definir una especie de método, aunque realmente es una propiedad computada. En el siguiente ejemplo creamos una instancia de la clase **Persona**, llamamos a la propiedad computada **nombreCompleto** y mostramos el resultado:

```
var persona = new Persona('Juan', 'Gutierrez');
var nombreCompletoPersona = persona.nombreCompleto;
console.log(nombreCompletoPersona);
```

En la variable **nombreCompletoPersona** disponemos del valor del nombre y apellidos concatenados. Para acceder al getter usamos **persona.nombreCompleto**.

Ejemplo de una clase Producto

```
class Producto{
    constructor(nombre, precio){
        this.nombre = nombre;
        this.precio = precio;
    }
    set actualizarPrecio(precio){
        this.precio = precio;
    }
    get precioFormateado(){
        return this.precio.toFixed(2);
    }
}
```

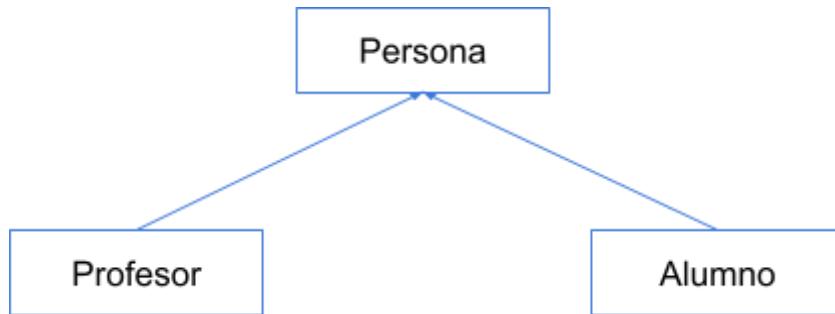
Creemos una instancia de Producto y llamemos a la propiedad computada **precioFormateado**:

```
var producto = new Producto("Coca", 12.5);
console.log(producto.precioFormateado);
```

El resultado sería:

```
$ "12.50"
```

Herencia



La herencia es una propiedad de la POO que permite que los objetos sean creados a partir de otros ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes. Es la relación entre una clase general y otra clase más específica. Es un mecanismo que nos permite crear clases derivadas a partir de clase base. Nos permite compartir automáticamente métodos y datos entre clases subclases y objetos. Por ejemplo una clase **Profesor** podría heredar de la clase **Persona** el método **hablar**. Dicho método también podría ser heredado por la clase **Alumno**. Si lo hicieramos sin la herencia, entonces deberíamos definir el método **hablar** en cada clase (Profesor y Alumno), duplicando código innecesariamente.

Veamos cómo se aplica este concepto en Javascript a través de los prototipos...

Prototipos

JavaScript provoca cierta confusión en desarrolladores con experiencia en lenguajes basados en clases (como Java o C++), por ser dinámico y no proporcionar una implementación de clases en sí mismo (la palabra clave `class` se introdujo en ES2015, pero sólo para endulzar la sintaxis, ya que JavaScript sigue estando basado en prototipos).

En lo que a herencia se refiere, JavaScript sólo tiene una estructura: **objetos**. Cada objeto tiene una propiedad privada (referida como su **Prototype**) que mantiene un enlace a otro objeto llamado su **prototipo**. Ese objeto prototipo tiene su propio prototipo, y así sucesivamente hasta que se alcanza un objeto cuyo prototipo es **null**. Por definición, **null** no tiene prototipo, y actúa como el enlace final de esta **cadena de prototipos**.

Casi todos los objetos en JavaScript son instancias de `Object` que se sitúa a la cabeza de la cadena de prototipos. Lo que queremos decir es que **un prototipo es un objeto del que otros objetos heredan propiedades**, y los objetos siempre heredan propiedades de algún objeto anterior, de este modo solo el objeto original y primigenio (**Object**) de javascript es el

único que no hereda de nadie.

Bien, de este modo podemos ver que:

- objetos creados por ejemplo de forma literal o new Object(), heredan directamente del objeto Object.prototype.
- objetos creados con new Date(), heredan de Date.prototype.
- Y así sucesivamente

Es el objeto **Object.prototype** el primer eslabón de la cadena, el padre u objeto primigenio, de este modo todos los objetos de Javascript heredan de él, ya sean arrays, fechas, funciones, etc.

Cómo definición de prototipo, podemos decir que **son un mecanismo por el cual un objeto hereda propiedades y métodos de un objeto padre**, entonces en Javascript la herencia funciona por prototipos, todo se hereda por medio de prototipos.

Prototype Chain(Cadena de prototipos)

Cada objeto tiene un prototipo, y este prototipo puede tener otro prototipo, así sucesivamente hasta encontrar a un prototipo que no tiene prototipo(el Object Prototype).

Vamos a crear un objeto usando una función constructora llamada **Persona**, y luego crearemos una instancia llamada **juan**, y si lo vemos en consola, vemos su propiedad nombre, pero también vemos algo llamado **prototype** que hace referencia al prototipo con el que fue creado nuestro objeto juan, el cual en este caso es la función constructora llamada Persona.

```
function Persona(nombre){  
    this.nombre = nombre;  
}  
var juan = new Persona('Juan');  
console.log(juan);
```

Si lo ejecutamos en el navegador deberíamos ver lo siguiente:

```
{...}  
nombre: "Juan"  
<prototype>: {...}  
constructor: function Persona()  
<prototype>: Object { ... }
```

En este ejemplo juan tiene como prototipo a Persona

Ahora, vamos a agregar un método a esa función constructora llamado **estudiar**, si

consultamos vemos que el método estudiar aparece y pertenece a la instancia de juan.

```
function Persona(nombre){  
    this.nombre = nombre;  
    this.estudiar = function(){  
        console.log("estudiando...");  
    }  
}  
  
var juan = new Persona('Juan');  
console.log(juan);
```

Si lo ejecutamos en el navegador deberíamos ver lo siguiente:

```
{...}  
estudiar: function estudiar()  
nombre: "Juan"  
<prototype>: {...}  
constructor: function Persona()  
<prototype>: Object { ... }
```

Acá llegamos a un punto importante, qué pasaría si creamos otra instancia de Persona, como se puede ver abajo, **por cada instancia se está copiando el método estudiar**, y es el mismo método, lo cual es algo ineficiente, no se reutiliza. Imaginémonos tener 1000 instancias de Persona, por cada una se copiaría el método estudiar afectando esto al rendimiento general del programa y a los recursos como la memoria ram.

```
var juan = new Persona('Juan');  
console.log(juan);  
  
var miguel = new Persona('Miguel');  
console.log(miguel);
```

El resultado:

```
{...}  
estudiar: function estudiar()  
nombre: "Juan"  
<prototype>: {...}  
constructor: function Persona()  
<prototype>: Object { ... }
```

```
{...}
estudiar: function estudiar()
nombre: "Miguel"
<prototype>: {...}
constructor: function Persona()
<prototype>: Object { ... }
```

Vemos que cada instancia juan y miguel tienen el método estudiar.

Esto sucede porque estamos utilizando una función constructora, y cada vez que se ejecuta crea las propiedades y los métodos dentro de cada instancia.

Moviendo el método al prototipo

Ya sabemos que cada instancia tiene un prototipo, en este caso fue el objeto con el que se creó, hablamos de la función constructora Animal(recordar las funciones también son objetos en JS).

Lo que deberíamos hacer es sacar el método fuera de la función constructora, y luego asignarlo por medio de **prototype**.

```
function Persona(nombre){
    this.nombre = nombre;
}

Persona.prototype.estudiar = function(){
    console.log("estudiando...");
}

var juan = new Persona('Juan');
console.log(juan);

var miguel = new Persona('Miguel');
console.log(miguel);
```

El resultado sería:

```
{...}
nombre: "Juan"
<prototype>: {...}
constructor: function Persona()
```

```
estudiar: function estudiar()
<prototype>: Object { ... }

{...}
nombre: "Miguel"
<prototype>: {...}
constructor: function Persona()
estudiar: function estudiar()
<prototype>: Object { ... }
```

Como vemos ahora el método no se está agregando en cada instancia de objeto, sino que se reutiliza del prototipo de Persona.

Prototipos y clases

Con las clases que nos trae ES6 este proceso de reutilizar se vuelve un poco más simple, porque aunque no son clases como las de otros lenguajes de programación, detrás de cámaras se encarga de hacer el proceso con los prototipos, por ejemplo si utilizamos la función constructora Persona, teníamos que agregarle el método de dormir, por medio del **prototype** nosotros manualmente, porque sino se repetiría en cada instancia.

Veamos un ejemplo de cómo se haría con una clase

```
class Persona{
    constructor(nombre){
        this.nombre = nombre;
    }

    estudiar(){
        console.log('estudiando...');
    }
}

var juan = new Persona('Juan');
console.log(juan);

var miguel = new Persona('Miguel');
console.log(miguel);
```

El resultado sería:

```
{...}
nombre: "Juan"
<prototype>: {...}
constructor: function Persona()
estudiar: function estudiar()
<prototype>: Object { ... }

{...}
nombre: "Miguel"
<prototype>: {...}
constructor: function Persona()
estudiar: function estudiar()
<prototype>: Object { ... }
```

Aplicando herencia

Ya vimos cómo funcionan los prototipos, y cómo funciona la herencia por medio de prototipos. Ahora vamos a crear un ejemplo, donde vamos a crear una función constructora llamada **Profesor**, la cual heredará una propiedad y un método de la función constructora **Persona**.

```
function Persona(nombre){
  this.nombre = nombre;
}

Persona.prototype.dormir = function(){
  console.log('durmiendo...');
}

// creamos una función constructora Profesor
function Profesor(nombre){
  // Llamamos al constructor padre, nos aseguramos (utilizando
  // Function#call)
  Persona.call(this, nombre);

  // propiedad propia
  this.enseñando = false;
}

// Creamos el objeto Profesor.prototype que hereda desde
Profesor.prototype
// Nota: Un error común es utilizar "new Persona()" para crear
Profesor.prototype
```

```
// Esto es incorrecto por varias razones, y no menos importante que no
Le estamos pasando nada
// a Persona desde el argumento "nombre". El Lugar correcto para Llamar
a Persona
// es arriba, donde Llamamos a Profesor.
Profesor.prototype = Object.create(Persona.prototype);

// Establecer La propiedad "constructor" para referencias a Profesor
Profesor.prototype.constructor = Profesor;

var profesor1 = new Profesor("Juan");

//Accediendo a una propiedad de Profesor
console.log("Está ladrando: "+profesor1.enseñando);

//Accediendo a una propiedad y un método de su padre
console.log("Nombre: "+profesor1.nombre);
profesor1.dormir();

/* Resultados:
"Está enseñando: false"
"Nombre: Juan"
"durmiendo..."*/

```

Aplicando herencia usando ES6

En ES6, simplemente tenemos que utilizar **extends** después de la declaración de la clase (`class`) e indicar el nombre de la clase de la cual queremos que herede todos los métodos y propiedades:

Veamos un ejemplo de cómo se haría con una clase

```
class Persona{
    constructor(nombre, apellido){
        this.nombre = nombre;
        this.apellido = apellido;
    }

    get nombreCompleto(){
        return this.nombre + ' ' + this.apellido;
    }
}
```

```
    }
}

class Profesor extends Persona{

    constructor(nombre, apellido, tipo){
        //llamamos al constructor de la clase Persona
        super(nombre, apellido);
        //inicializamos las propiedades propias de la clase Profesor
        this.tipo = tipo;
    }

    get tipoFormateado(){
        return 'Profesor: ' + this.tipo;
    }
}

var profe1 = new Profesor('Juan','Subelza','permanente');
console.log(profe1.nombreCompleto);
console.log(profe1.tipoFormateado);

var profe2 = new Profesor('Miguel','Farias','Iterino');
console.log(profe2.nombreCompleto);
console.log(profe2.tipoFormateado);
```

El resultado del ejemplo anterior sería el siguiente:

```
Juan Subelza
Profesor: permanente
Miguel Farias
Profesor: Iterino
```

En el ejemplo anterior vimos que usamos `super` para llamar al constructor padre de la clase **Profesor**. `Super` debe ubicarse siempre como la primera línea de código después de la declaración del constructor de la clase hija. Si lo vemos de otra manera sería después de que la palabra clave `this` sea usada. La palabra clave `super` también puede utilizarse para llamar otras funciones del objeto padre. `Super` también recibe los parámetros que necesita recibir el constructor de la clase padre.

Encapsulación

En el ejemplo anterior, Profesor no tiene que saber cómo se aplica el método nombreCompleto() de la clase Persona, pero, sin embargo, puede utilizar ese método. La clase Persona no tiene que definir explícitamente ese método, a menos que queramos cambiarlo. Esto se denomina **la encapsulación**, por medio de la cual cada clase hereda los métodos de su elemento primario y sólo tiene que definir las cosas que desea cambiar.

Abstracción

Es un mecanismo que permite modelar la parte actual del problema de trabajo. Esto se puede lograr por herencia (especialización) o por composición. JavaScript logra la especialización por herencia y por composición al permitir que las instancias de clases sean los valores de los atributos de otros objetos.

La clase Function de JavaScript hereda de la clase de Object (esto demuestra la especialización del modelo) y la propiedad Function.prototype es un ejemplo de Objeto (esto demuestra la composición);

Polimorfismo

En POO se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

En algunos lenguajes, el término polimorfismo es también conocido como 'Sobrecarga de parámetros' ya que las características de los objetos permiten aceptar distintos parámetros para un mismo método (diferentes implementaciones) generalmente con comportamientos distintos e independientes para cada una de ellas.

Al igual que todos los métodos y propiedades están definidas dentro de la propiedad prototipo, las diferentes clases pueden definir métodos con el mismo nombre. Los métodos están en el ámbito de la clase en que están definidos. Esto sólo es verdadero cuando las dos clases no tienen una relación primario-secundario (cuando uno no hereda del otro en una cadena de herencia).

Veamos un ejemplo:

```
class Persona{  
    constructor(nombre, apellido){  
        this.nombre = nombre;  
        this.apellido = apellido;
```

```
}

get nombreCompleto(){
    return this.nombre + ' ' + this.apellido;
}

cambiarNombreCompleto(nombre){
    this.nombre = nombre;
}

cambiarNombreCompleto(nombre, apellido){
    this.nombre = nombre;
    this.apellido = apellido;
}

}

class Profesor extends Persona{
    constructor(nombre, apellido, tipo){
        super(nombre, apellido);
        this.tipo = tipo;
    }
    get tipoFormateado(){
        return 'Profesor: ' + this.tipo;
    }
}

var profe1 = new Profesor('Juan','Subelza','permanente');
console.log(profe1.nombreCompleto);
console.log(profe1.tipoFormateado);
profe1.cambiarNombreCompleto('Miguel');
console.log(profe1.nombreCompleto);
profe1.cambiarNombreCompleto('Jorge','Fernandez');
console.log(profe1.nombreCompleto);
```

El resultado del ejemplo anterior es:

```
Juan Subelza
Profesor: permanente
Miguel Subelza
Jorge Fernandez
```

Lo que hicimos básicamente es definir dos veces el método `cambiarNombreCompleto` en la clase **Persona** pero con distintos parámetros . El primero sólo recibe el **nombre**, la segunda declaración del método recibe **nombre ya apellido**, y es lo que normalmente se

denomina **sobrecarga de métodos** y es a lo que se refiere el concepto de polimorfismo.

Javascript en el navegador

Javascript puede ser utilizado en [diferentes entornos](#), pero su entorno más habitual es el navegador

El código Javascript de una pagina tiene acceso a unos cuantos objetos. Estos objetos los podemos agrupar en:

- Objetos que tienen relación con la pagina cargada(`eldocument`). Estos objetos conforman el [Document Object Model \(DOM\)](#)
- Objetos que tienen que ver con cosas que están fuera de la pagina(`laventanadelnavegador` y la pantalla). Estos objetos conforman el [Browser Object Model \(BOM\)](#)

El **DOM** es un standard y tiene varias [versiones](#) (llamadas levels). La mayoría de los navegadores implementan casi por completo el DOM Level 1.

El **BOM** no es un standard, así que algunos objetos están soportados por la mayoría de navegadores y otros solo por algunos.

Detectar funcionalidades

Debido a estas diferencia entre navegadores surge la necesidad de averiguar (desde código JS) que características soporta nuestro navegador (DOM y BOM)

Una solución sería la llamada [Browser Sniffing](#) que consiste en detectar el navegador que estamos utilizando.

Esta técnica no se recomienda por:

- Hay demasiados navegadores para controlar
- Difícil de mantener (surgen nuevas versiones y nuevos navegadores)
- El parseo de cadenas puede ser complicado y no es fiable del todo

Ejemplo:

```
if (navigator.userAgent.indexOf('MSIE') !== -1) {  
    //el navegador es internet explorer  
} else {  
    //el navegador no es internet explorer  
}
```

La mejor solución para detectar funcionalidades de nuestro navegador es hacer [Feature Sniffing](#), es decir chequear la existencia del objeto (método, array o propiedad) que queremos utilizar.

```
if (typeof window.addEventListener === 'function') {  
    // La característica está soportada  
} else {  
    // mm, esta característica no está soportada, piensa en otra forma  
    // para implementar esto  
}
```

BOM

El **BOM (Browser Object Model)** lo conforman todos los objetos que están fuera del documento cargado (document) y forman parte del objeto window.

El objeto **window** además de servir de contenedor de las variables globales y de ofrecer los métodos nativos de JS (window.parseInt), contiene información sobre el entorno del navegador (frame, iframe, popup, ventana o pestaña).

Propiedades de window

Algunos de los **objetos** que tenemos disponibles en window son:

window.navigator

Es un objeto que contiene información sobre el navegador

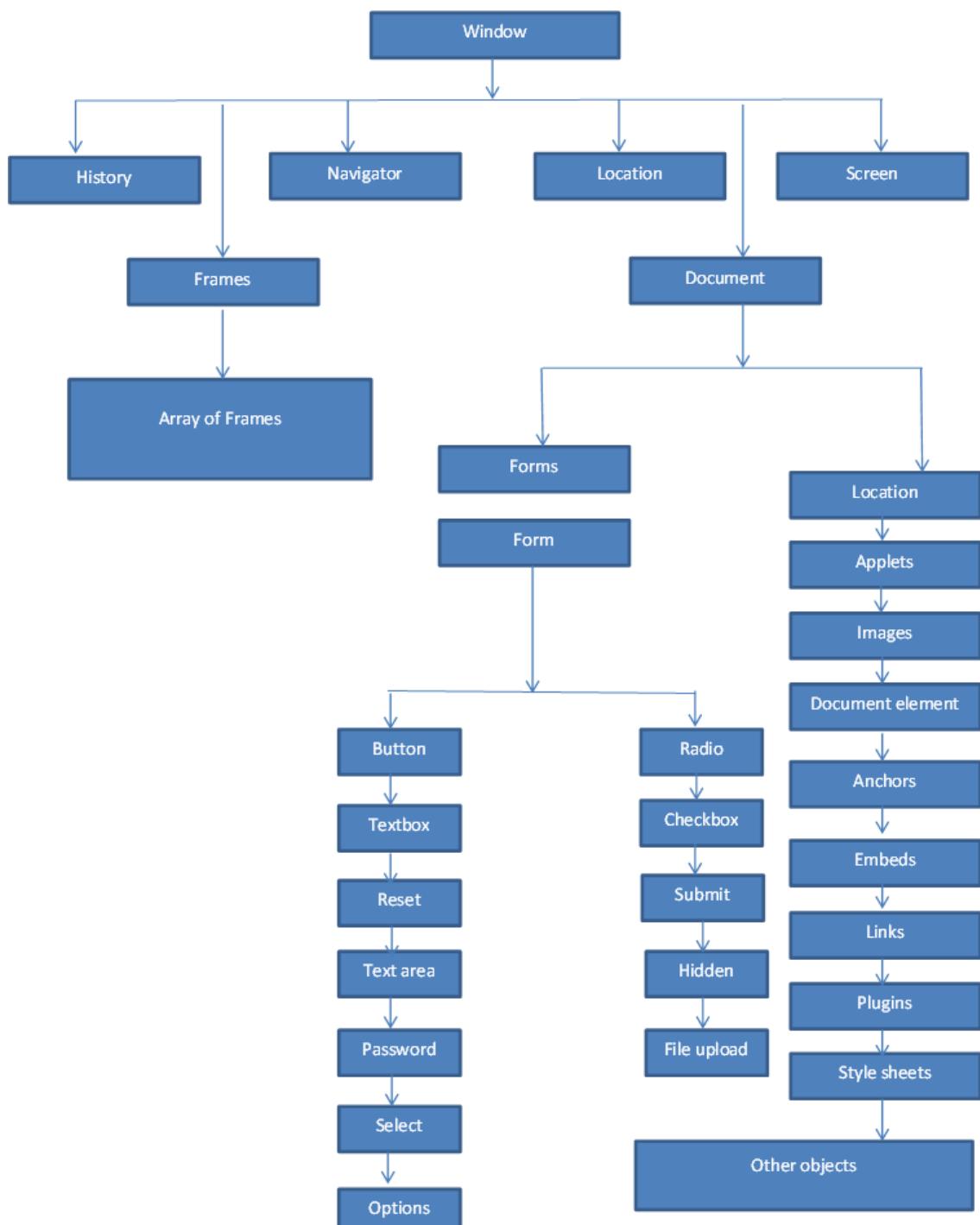
```
> window.navigator.userAgent  
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36  
(KHTML, like Gecko) Chrome/76.0.3809.100 Safari/537.36"
```

window.location

Es un objeto que contiene info (y métodos) sobre la URL actual

```
> window.location.href = 'https://rollingcodeschool.com'  
> location.href = 'https://rollingcodeschool.com'  
> location = 'https://rollingcodeschool.com'
```

```
> location.assign('https://rollingcodeschool.com')
> location.reload()
> window.location.href = window.location.href
> location = location
```



BOM (Browser Object Model)

window.history

Es un objeto que contiene el historial de páginas visitadas y tiene métodos para movernos en él (sin poder ver las URL's).

```
> window.history.length
5
> history.forward()
> history.back()
> history.go(-2);
```

window.frames

Es una colección de todos los frames que tenemos en la página.

Cada frame tendrá su propio objeto window.

Podemos utilizar **parent** para acceder desde el frame hijo al padre.

Con la propiedad **top** accedemos a la página que está por encima de todos los frames.

Podemos acceder a un frame concreto por su nombre.

```
<iframe name="myframe" src="about:blank" />
> window.frames[0]
> frames[0].window.location.reload()
> frames[0].parent === window
true
> window.frames[0].window.top === window
true
> self === window
true
> window.frames['myframe'] === window.frames[0]
true
```

window.screen

Ofrece info sobre la pantalla (general, fuera del browser)

```
> window.screen.colorDepth
24
> screen.width
1440
```

```
> screen.availWidth  
1440  
> screen.height  
900  
> screen.availHeight  
803
```

Métodos de window

Algunos de los métodos que tenemos disponibles en window son:

[window.open\(\)](#), [window.close\(\)](#)

Nos permiten abrir (y cerrar) nuevas ventanas (popups)

window.open() devuelve una referencia a la ventana creada (si devuelve false es que no la ha podido crear - popups blocked). No se recomienda su uso.

```
> var win = window.open('https://rollingcodeschool.com',  
'nombreVentana', 'width=300,height=300,resizable=yes');  
> win.close()
```

[window.moveTo\(\)](#), [window.moveBy\(\)](#), [window.resizeTo\(\)](#), [window.resizeBy\(\)](#)

Nos permiten mover y redimensionar las ventanas No se recomienda su uso.

```
> window.moveTo(100, 100)  
> window.moveBy(10, -10)  
> window.resizeTo(300, 300)  
> window.resizeBy(20, 10)
```

[window.alert\(\)](#), [window.prompt\(\)](#), [window.confirm\(\)](#)

Nos permiten interactuar con el usuario a través de mensajes del sistema.

Ejemplo: Mostrar un mensaje de confirmación al usuario

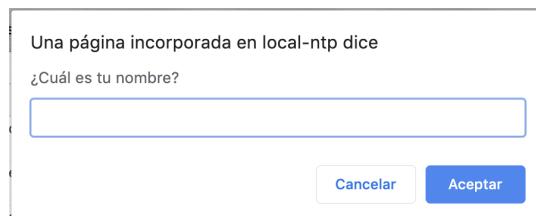
```
if (confirm('¿Estás seguro que deseas eliminar el producto?')) {
```

```
// eliminar
else{
    // abortar acción
}
```



Ejemplo: Preguntar el nombre del usuario:

```
> var name = prompt('¿Cuál es tu nombre?');
console.log(name);
```



[window.setTimeout\(\)](#), [window.setInterval\(\)](#)

Nos permiten ejecutar código después de un intervalo de tiempo (y en su caso, repetirlo).

```
> function boo(){alert('Boo!')}
> setTimeout(boo, 2000);
> var id = setTimeout(boo, 2000);
> clearTimeout(id);
> function boo() { console.log('boo') };
> var id = setInterval( boo, 2000 );
boo
boo
boo
> clearInterval(id)
```

El método **setTimeout()** permite ejecutar código después de un cierto tiempo. Recibe 2 parámetros, una **función** y el **tiempo** en milisegundos a partir del cual ejecutará la función pasada como parámetro.

El método **setInterval()** permite ejecutar código cada x tiempo. Recibe 2 parámetros, una **función** y el **tiempo** en milisegundos.

`setTimeout()` y `setInterval()` retornan un id, el cual permite **detener** la ejecución a través del método **clearTimeout()** y **clearInterval()** respectivamente que recibe dicho id.

Veamos un ejemplo de una simulación de lanzamiento de una nave espacial:

```
var contador = 6; //segundos para el Lanzamiento
var idProceso; //id del proceso a limpiar una vez Lanzada La nave

function iniciarLanzamiento(){
    idProceso = setInterval(contadorReversa,1000); //Llamamos a
    contadorReversa cada 1seg
}

function contadorReversa(){
    if(contador == 0){
        console.log('lanzamiento iniciado');
        clearInterval(idProceso);
    }else{
        console.log('iniciando lanzamiento en ' + contador);
        contador--;
    }
}

function calentarMotores(){
    console.log("Calentando motores...");
    setTimeout(iniciarLanzamiento,5000); //iniciamos Lanzamiento en 5
    segundos
}

calentarMotores(); //iniciamos precalentamiento de motores
```

El resultado sería:

```
Calentando motores...
iniciando lanzamiento en 6
iniciando lanzamiento en 5
iniciando lanzamiento en 4
iniciando lanzamiento en 3
```

```
iniciando lanzamiento en 2
iniciando lanzamiento en 1
lanzamiento iniciado
```

El objeto document

`window.document` es un objeto del BOM con información sobre el documento html actual. Todos los métodos y propiedades que estan dentro de `window.document` pertenecen a la categoría de objetos DOM.

DOM

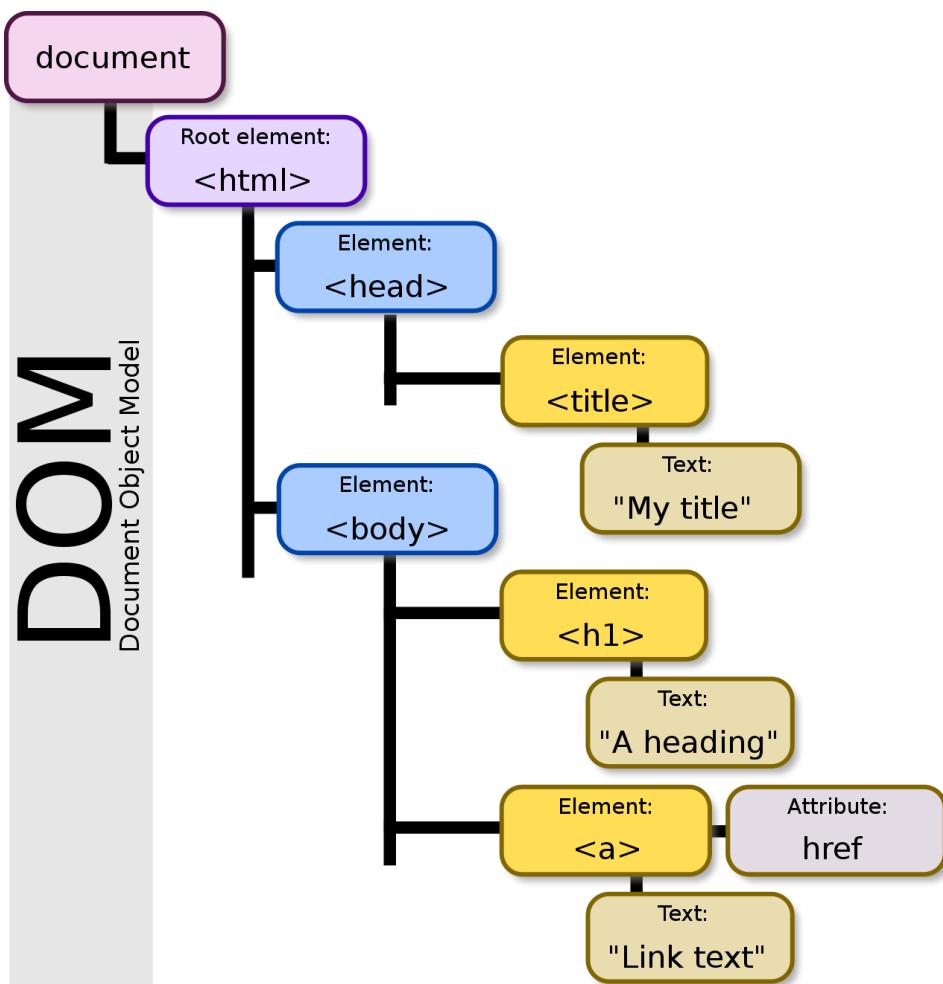
El **DOM (Document Object Model)** es una forma de representar un documento HTML (o XML) como un árbol de nodos.

Utilizando los métodos y propiedades del DOM podremos acceder a los elementos de la página, modificarlo, eliminarlos o añadir nuevos.

Un documento html típico se ve como el código de a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>My Title</title>
</head>
<body>
  <h1>A Heading</h1>
  <a href="https://rollingcodeschool.com">Link Text</a>
</body>
</html>
```

El árbol DOM de ese documento es como se muestra en la siguiente imagen. Es común denominarlo árbol porque tiene esa estructura jerárquica.



DOM (Document Object Model)

Accediendo a los nodos

```

<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
  
```

El nodo [document](#) nos da acceso al documento (es el punto de partida). Todos los nodos tienen las propiedades:

- **nodeType**: Hay 12 tipos de nodos representados por números (1=element, 2=attribute, 3=text, ...)
- **nodeName**: Para tags HTML es el nombre del tag y para nodos texto es #text

-
- **nodeValue**: Para nodos de texto el valor será el texto

El nodo **documentElement** es el nodo raíz. Para documentos HTML es el tag <html>

Veamos un ejemplo:

```
> document.documentElement
<html>
> document.documentElement.nodeType
1
> document.documentElement.nodeName
"HTML"
> document.documentElement.tagName
"HTML"
```

Cada nodo puede tener nodos-hijo:

- **hasChildNodes()** : Este método devolverá **true** si el nodo tiene nodos-hijo
- **childNodes**: Devuelve en un array los nodos-hijo de un elemento. Al ser un array podemos saber el número de nodos-hijo con **childNodes.length**
- **parentNode**: Nos da el nodo-padre de un nodo-hijo

```
> document.documentElement.hasChildNodes()
True
> document.documentElement.childNodes.length
2
> document.documentElement.childNodes[0]
<head>
> document.documentElement.childNodes[1]
<body>
> document.documentElement.childNodes[1].parentNode
<html>
> var bd = document.documentElement.childNodes[1];
> bd.childNodes.length
9
```

Podemos chequear la existencia de atributos y acceder a ellos:

- **hasAttributes()**: Devuelve true si el elemento tiene atributos
- **getAttribute()**: Devuelve el contenido de un atributo

```
> bd.childNodes[1]
<p class="opener">
> bd.childNodes[1].hasAttributes()
True
> bd.childNodes[1].attributes.length
1
> bd.childNodes[1].attributes[0].nodeName
"class"
> bd.childNodes[1].attributes[0].nodeValue
"opener"
> bd.childNodes[1].attributes['class'].nodeValue
"opener"
> bd.childNodes[1].getAttribute('class')
"opener"
```

Acceder al contenido de un tag

Podemos acceder al contenido de un tag:

- **textContent**: Esta propiedad nos da el texto plano dentro de una etiqueta En IE no existe esta propiedad (hay que usar innerText)
- **innerHTML**: Esta propiedad nos da el contenido (en HTML) de un tag

```
> bd.childNodes[1].nodeName
"P"
> bg.childNodes[1].textContent
"first paragraph"
> bd.childNodes[1].innerHTML
"first paragraph"
> bd.childNodes[3].innerHTML
"<em>second</em> paragraph"
> bd.childNodes[3].textContent
"second paragraph"
> bd.childNodes[1].childNodes.length
1
> bd.childNodes[1].childNodes[0].nodeName
"#text"
> bd.childNodes[1].childNodes[0].nodeValue
"first paragraph"
```

Acceso directo a tags

Podemos acceder directamente a algunos elementos sin necesidad de recorrer todo el árbol a través de los siguientes métodos de document:

- **getElementsByName()**: Nos devuelve un array con todos los elementos con el tag que se le pasa por parámetro
- **getElementsByName()**: Nos devuelve un array con todos los elementos con el name que se le pasa por parámetro
- **getElementById()**: No devuelve el elemento con el id que se le pasa por parámetro. El id corresponde al atributo id que se coloca en una etiqueta html.

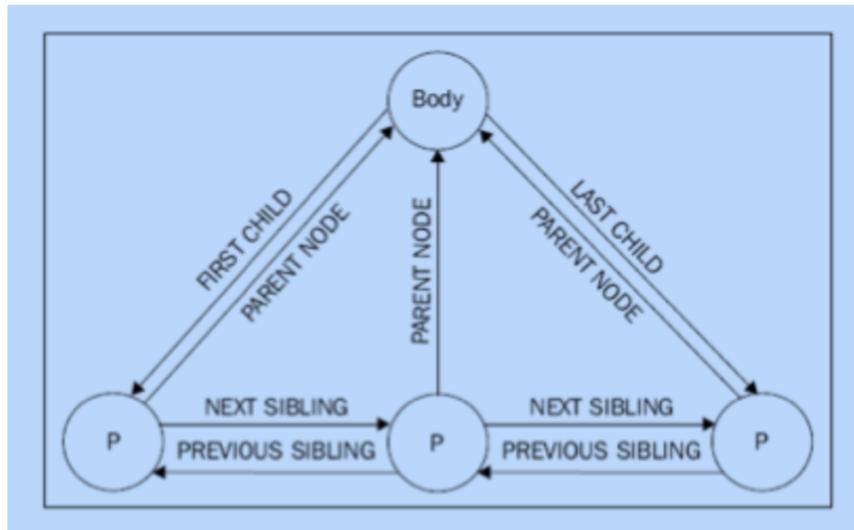
Veamos un ejemplo:

```
> document.getElementsByTagName('p').length
3
> document.getElementsByTagName('p')[0]
<p class="opener">
> document.getElementsByTagName('p')[0].innerHTML
"first paragraph"
> document.getElementsByTagName('p')[2]
<p id="closer">
> document.getElementsByTagName('p')[2].id
"closer"
> document.getElementsByTagName('p')[0].className
"opener"
> document.getElementById('closer')
<p id="closer">
```

Parent y Childs

Desde un nodo también podemos acceder a sus hermanos y al primero y último de sus hijos.

- **nextSibling**: Nos devuelve el siguiente hermano
- **previousSibling**: Nos devuelve el anterior hermano
- **firstChild**: Nos devuelve el primer hijo
- **lastChild**: Nos devuelve el último hijo



Parents y childs

Veamos un ejemplo:

```
> var para = document.getElementById('closer')
> para.nextSibling
"\n"
> para.previousSibling
"\n"
> para.previousSibling.previousSibling
<p>
> para.previousSibling.previousSibling.previousSibling
"\n"
> para.previousSibling.previousSibling.nextSibling.nextSibling
<p id="closer">
> document.body.previousSibling
<head>
> document.body.firstChild
"\n"
> document.body.lastChild
"\n"
> document.body.lastChild.previousSibling
Comment length=21 nodeName=#comment
> document.body.lastChild.previousSibling.nodeValue
" and that's about it "
```

Modificando los nodos

Para cambiar el contenido de una etiqueta, tenemos que cambiar el contenido de **innerHTML**.

Veamos un ejemplo:

```
> var my = document.getElementById('closer');
> my.innerHTML = '<em>my</em> final';
> my.firstChild
4 <em>
> my.firstChild.firstChild
"my"
> my.firstChild.firstChild.nodeValue = 'your';
"your"
```

Los elementos tienen la propiedad **style** que podemos utilizar para modificar sus estilos.

```
> my.style.border = "1px solid red";
"1px solid red"
```

Además podemos modificar los atributos existan previamente o no.

Por ejemplo:

```
> my.align = "right";
"right"
> my.name
> my.name = 'myname';
"myname"
> my.id
"closer"
> my.id = 'further'
"further"
```

Creando y eliminando nodos

Para crear nuevos elementos podemos utilizar los métodos **createElement** y **createTextNode**. Una vez creados los podemos añadir al DOM con **appendChild**

```
> var myp = document.createElement('p');
> myp.innerHTML = 'yet another';
"yet another"
> myp.style
CSSStyleDeclaration length=0
```

```
> myp.style.border = '2px dotted blue'  
"2px dotted blue"  
> document.body.appendChild(myp)  
<p style="border: 2px dotted blue;">
```

También podemos copiar elementos existentes con **cloneNode()**
cloneNode acepta un parámetro booleano (true copiará el nodo con todos sus hijos y false solo el nodo).

```
> var el = document.getElementsByTagName('p')[1];  
<p><em>second</em> paragraph</p>  
> document.body.appendChild(el.cloneNode(false))  
> document.body.appendChild(document.createElement('p'));  
> document.body.appendChild(el.cloneNode(true))
```

Con **insertBefore()** podemos especificar el elemento delante del cual queremos insertar el nuestro.

```
document.body.insertBefore(  
  document.createTextNode('boo!'),  
  document.body.firstChild  
)
```

Para eliminar nodos del DOM podemos utilizar **removeChild()** o **replaceChild()**
removeChild() elimina el elemento y replaceChild() lo sustituye por otro que se le pasa como parámetro.

Tanto **replaceChild()** como **removeChild()** devuelven el nodo eliminado.

Veamos un ejemplo:

```
> var myp = document.getElementsByTagName('p')[1];  
> var removed = document.body.removeChild(myp);  
> removed  
<p>  
> remove.firstChild  
<em>  
> var p = document.getElementsByTagName('p')[1];  
> p  
<p id="closer">  
> var replaced = document.body.replaceChild(removed, p);  
> replaced  
<p id="closer">
```

Objetos DOM sólo de HTML

En el DOM tenemos disponibles una serie de selectores directos y de colecciones exclusivos de HTML (no XML):

- **document.body**: `document.getElementsByTagName('body')[0]`
 - **document.images**: `document.getElementsByTagName('img')`
 - **document.applets**: `document.getElementsByTagName('applet')`
 - **document.links**: Nos devuelve un array con todos los links con atributo href
 - **document.anchors**: Nos devuelve un array con todos los links con atributo name
 - **document.forms**: `document.getElementsByTagName('form')`
- Podemos acceder a los elementos del form (inputs, buttons) con la propiedad **elements**

Veamos un ejemplo para obtener un formulario:

```
> document.forms[0]
> document.getElementsByName( 'forms' )[0]
> document.forms[0].elements[0]
> document.forms[0].elements[0].value = 'me@example.org'
"me@example.org"
> document.forms[0].elements[0].disabled = true;
> document.forms[0].elements['search']; // array notation
> document.forms[0].elements.search; // object property
```

También tenemos disponible el método **document.write()** para crear elementos directamente en el DOM.

```
document.write("<h1>Hello World!</h1><p>Have a nice day!</p>");
```

Al usar **document.write()** después de que un documento HTML ha sido cargado completamente, este eliminará todo el contenido existente en el documento.

Por ejemplo:

```
// Esto debería ser evitado:
function myFunction() {
  document.write("Hello World!");
}
```

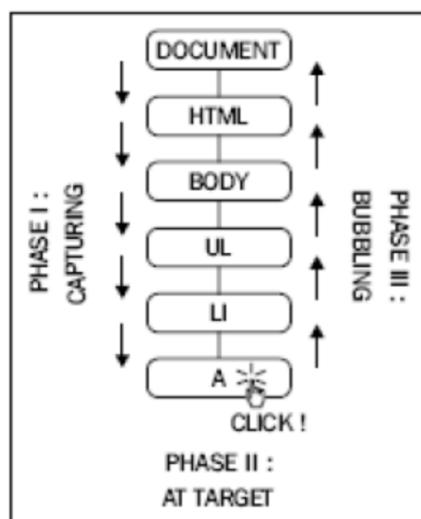
En el ejemplo anterior vemos que hemos puesto **document.write()** dentro de una función. Cuando la función es invocada, todos los elementos HTML serán sobreescritos y reemplazados con el texto 'Hello World'. Por eso es que **document.write()** se usa sólo para pruebas y se debe **evitar su uso**.

Algunas propiedades del objeto document son:

- **document.cookies**: Contiene una cadena de texto con las cookies asociadas al documento
- **document.title**: Permite cambiar el título de la página que aparece en el navegador. Esto no cambia el contenido del tag title
- **document.referrer**: Contiene la URL desde donde hemos llegado a la página
- **document.domain**: Contiene el dominio de la pagina

Eventos

Cada acción (click, change, select, etc) que ocurre en el navegador es comunicada (a quien quiera escuchar) en forma de **evento**. Desde Javascript podemos escuchar estos eventos y engancharle una función (**event handler**) que se ejecutará cuando ocurra este evento.



Cuando hacemos click en un link (a), tambien hacemos click en su contenedor (li,ul, etc), en el body y en última instancia en el document. Esto es lo que se llama la **propagación del evento**.

La [especificación de eventos DOM Level 2](#) define que el evento se propaga en 3 fases: **Capturing, Target y Bubbling**

- **Capturing**: El click ocurre en el document y va pasando por todos los elementos hasta llegar al link (a)

-
- **Bubbling:** El click ocurre en el link (a) y va emergiendo hasta el document.

Capturar eventos

```
function callback(evt){  
    // comprobación previa  
    evt = evt || window.event;  
    var target = (typeof evt.target !== 'undefined') ? evt.target :  
    evt.srcElement;  
    // se imprime el nombre del nodo desde donde se ejecutó el evento  
    console.log(target.nodeName);  
}  
  
// comenzamos a escuchar por eventos click en el documento  
if (document.addEventListener){ // para firefox, opera, chrome  
    document.addEventListener('click', callback, false);  
} else if (document.attachEvent){ // para internet explorer  
    document.attachEvent('onclick', callback);  
} else{  
    document.onclick = callback;  
}
```

Modelo tradicional

Podemos capturar eventos con el [modelo tradicional](#).

Este modelo consiste en asignar una función a la propiedad onclick, onchange,... del elemento del DOM.

```
<button onClick="myFunction()">Dame click</button>  
<button onClick="document.bgColor='lightgreen'">Dame click</button>  
<button onClick="getElementById('demo').innerHTML=Date()">¿Qué hora  
es?</button>  
<a href="#" onClick="alert('Hola a todos!');">Saludar</a>
```

```
window.onclick = myFunction;  
// si el usuario clickea en la ventana, se pone el color de fondo del  
//body a amarillo.  
function myFunction() {  
    document.getElementsByTagName("body")[0].style.backgroundColor =  
    "yellow";
```

```
}
```

Con este metodo sólo podemos asignar UNA funcion a cada evento. Este método funciona igual en TODOS los navegadores.

Modelo avanzado

También podemos capturar eventos con el [modelo avanzado](#).

Con este método podemos asignar varias funciones a un mismo evento

Este modelo se aplica distinto según el navegador

Para enganchar/desenganchar una función a un evento con este modelo se utiliza:

- **addEventListener** y **removeEventListener** en Firefox, Opera y Safari (W3C way) Le pasamos 3 parámetros:
 - **Tipo de Evento**: click,change,...
 - **Función a ejecutar** (handler, callback) : Recibe un objeto e con info sobre el evento. En e.target tenemos el elemento que lanzó el evento
 - **¿Utilizo Capturing?**: Poniéndolo a false utilizariamos sólo Bubbling
- **attachEvent** y **detachEvent** en IE (Microsoft way) Le pasamos 2 parámetros:
 - **Tipo de Evento**: onclick, onchange,...
 - **Función a ejecutar** (handler,callback): Para acceder a la info del evento hay que mirar el objeto global window.event. En event.srcElement tenemos el elemento que lanzó el evento

Veamos un ejemplo:

```
<button id="button1">Dame click</button>
```

```
let button = document.getElementById('button1');
button.addEventListener('click', enviarDatos, false);
button.addEventListener('click', calcularDatos, false);
button.addEventListener('click', refrescarPantalla, false);
function enviarDatos(e){
    console.log("enviar datos..." + e.target);
}

function calcularDatos(e){
    console.log("calcular datos..." + e.target);
}
```

```
function refrescarPantalla(e){  
    console.log("refrescar pantalla..." + e.target);  
}
```

En este caso estamos registrando 3 funciones manejadores de eventos para el elemento **button1**. Podemos registrar tantas funciones manejadoras de eventos como nosotros queramos. Vemos que estas funciones se ejecutarán cuando hagamos click en el botón. La W3C model no establece criterio u orden de cual de las funciones se ejecutará primero. Entonces no podemos presumir que la función **enviarDatos()** se ejecute antes que **calcularDatos()**.

Eliminar manejadores de eventos

Podemos eliminar un manejador de eventos(event handler) registrado previamente usando **removeEventListener**.

```
button.removeEventListener('click', refrescarPantalla, false);
```

Al eliminar un manejador de eventos, la función asociada al mismo dejará de ejecutarse al hacer click sobre el botón del ejemplo.

Detener el flujo de eventos

Algunos elementos tienen un comportamiento por defecto (por ejemplo al hacer click sobre un link nos lleva a su URL). O al hacer click sobre un botón tipo submit nos enviará la información del formulario a la url configurada en el atributo action.

Esta acción por defecto se ejecuta al final (si tenemos otras funciones asignadas al evento)

Para desactivar la acción por defecto utilizamos el metodo **e.preventDefault()**. En IE pondremos a false la propiedad **returnValue** de **window.event**.

Podemos detener la propagación del evento con el método **e.stopPropagation()**. En IE pondremos a true la propiedad **cancelBubble** de **window.event**

Cuando la función asignada al evento devuelve false se aplica automáticamente **e.preventDefault()** y **e.stopPropagation()**.

Veamos un ejemplo:

```
<a href="http://google.com" id="enlace">Ir a google</a>
```

```
let enlace = document.getElementById('enlace');
enlace.addEventListener('click', validar, false);

function validar(e){
    e.preventDefault();
    e.stopPropagation();
    // si ponemos return false ya no hace falta
    // poner preventDefault y stopPropagation
    // return false;

    // ejecutar acciones de validación. Esto es a modo representativo
    // del ejemplo
    console.log("ejecutando acciones de validación");
}
```

En el ejemplo anterior cuando el usuario haga click en el link con id **enlace**, se esperaría que el navegador lo redirecciona a google , sin embargo como se llama a **preventDefault** y **stopPropagation**, la acción se detiene. Recordemos que preventDefault y stopPropagation se puede reemplazar con la expresión **return false** sencillamente.

Delegación de eventos

Aprovechando el bubbling y la detección del target podemos optimizar (en algunos casos) nuestra gestión de eventos con la delegación de eventos.

Para el caso en que tengamos que capturar los eventos de muchos elementos (por ejemplo los clicks en celdas de una tabla), podemos capturar el evento de su contenedor (la tabla) y detectar luego cual de sus hijos (qué celda) provocó el evento,

Las principales ventajas de este sistema son:

- Hay muchas menos definiciones de eventos: **menos espacio en memoria y mayor performance**.
- No hay que re-capturar los eventos de los elementos añadidos dinámicamente

Listado de manejadores de eventos

La siguiente tabla muestra los manejadores de eventos que pueden utilizarse en JavaScript, la versión a partir de la cual están soportados y su significado.

Manejador	Versión	Se produce cuando...
-----------	---------	----------------------

onAbort	1.1	El usuario interrumpe la carga de una imagen
onBlur	1.0	Un elemento de formulario, una ventana o un marco pierden el foco
onChange	1.0 (1.1 para FileUpload)	El valor de un campo de formulario cambia
onClick	1.0	Se hace click en un objeto o formulario
onDblClick	1.2 (no en Mac)	Se hace click doble en un objeto o formulario
onDragDrop	1.2	El usuario arrastra y suelta un objeto en la ventana
onError	1.1	La carga de un documento o imagen produce un error
onFocus	1.1 (1.2 para Layer)	Una ventana, marco o elemento de formulario recibe el foco
onKeyDown	1.2	El usuario pulsa una tecla
onKeyPress	1.2	El usuario mantiene pulsada una tecla
onKeyUp	1.2	El usuario libera una tecla
onLoad	1.0 (1.1 para image)	El navegador termina la carga de una ventana
onMouseDown	1.2	El usuario pulsa un botón del ratón
onMouseMove	1.2	El usuario mueve el puntero
onMouseOut	1.1	El puntero abando una área o enlace

onMouseOver	1.0 (1.1 para area)	El puntero entra en una área o imagen
onMouseUp	1.2	El usuario libera un botón del ratón
onMove	1.2	Se mueve una ventana o un marco
onReset	1.1	El usuario limpia un formulario
onResize	1.2	Se cambia el tamaño de una ventana o marco
onSelect	1.0	Se selecciona el texto del campo texto o área de texto de un formulario
onSubmit	1.0	El usuario envía un formulario
onUnload	1.0	El usuario abandona una página

Veamos un ejemplo:

```
<input type="text" onChange="compruebaCampo(this)">
```

En este ejemplo, `compruebaCampo()` es una función JavaScript definida en alguna parte del documento HTML (habitualmente en la cabecera del mismo). El identificador `this` es una palabra propia del lenguaje, y se refiere al objeto desde el cual se efectúa la llamada a la función (en este caso, el campo del formulario).

La siguiente tabla muestra los eventos que pueden utilizarse con los objetos del modelo de objetos JavaScript del Navegador.

Manejador de evento	Objetos para los que está definido
onAbort	Image

onBlur	Button, Checkbox, FileUpload, Layer, Password, Radio, Reset, Select, Submit, Text, Textarea, window
onChange	FileUpload, Select, Text, Textarea
onClick	Button, document, Checkbox, Link, Radio, Reset, Submit
onDbClick	document, Link
onDragDrop	window
onError	Image, window
onFocus	Button, Checkbox, FileUpload, Layer, Password, Radio, Reset, Select, Submit, Text, Textarea, window
onKeyDown	document, Image, Link, Textarea
onKeyPress	document, Image, Link, Textarea
onKeyUp	document, Image, Link, Textarea
onLoad	Image, Layer, window
onMouseDown	Button, document, Link
onMouseMove	Ninguno (debe asociarse a uno)
onMouseOut	Layer, Link
onMouseOver	Layer, Link
onMouseUp	Button, document, Link
onMove	window
onReset	Form
onResize	window
onSelect	Text, Textarea

onSubmit	Form
onUnload	window

Métodos de evento disponibles en JavaScript

Los siguientes métodos de evento pueden utilizarse en JavaScript:

Métodos de evento	Función que realizan
blur()	Elimina el foco del objeto desde el que se llame
click()	Simula la realización de un click sobre el objeto desde el que se llame
focus()	Lleva el foco al objeto desde el que se llame
select()	Selecciona el área de texto del campo desde el que se llame
submit()	Realiza el envío del formulario desde el que se llame

Ejemplo:

```
function validarCampoVacio(campo) {
    if(!campo.value){
        alert("¡Introduzca un valor!")
        campo.focus();
    }
}
```

```
<form method="post">
    <input type="text" name="campo" onBlur="validarCampoVacio(this)">
</form>
```

En el ejemplo anterior cuando el campo pierde el foco (**onBlur**), se llama a la función **validarCampoVacio**, que valida si el usuario a ingresado algún valor en el mismo. Si no lo ha hecho lanza una alerta con un mensaje y pone el foco nuevamente en el campo usando la

función **focus()** del objeto campo.

Eventos onLoad y onUnLoad

Se usan como atributos del tag **<body>** de html.

Ejemplo:

```
<body onLoad="Hola()" onUnload="Adios()">
```

La función Hola() se ejecutará antes de que se cargue la página y la función Adios() al abandonarla.

```
<body onLoad="alert('¡Bienvenido a mi página!')"  
onUnload="alert('¡Vuelva pronto!')">  
...  
</body>
```

En este otro ejemplo se utilizan funciones:

```
var name = ""  
function Hola() {  
    nombre = prompt('Introduzca su nombre:', '')  
    alert('¡Hola ' + nombre + '!')  
}  
function Adios() {  
    alert('¡Adios ' + nombre + '!')  
}
```

```
<body onLoad="Hola()" onUnload="Adios()">  
...  
</body>
```

JSON

JSON es el acrónimo de **JavaScript Object Notation**. Notación de objeto JavaScript. Es un objeto JavaScript pero con algunos detalles de implementación que nos permitirán serializarlo para poder utilizarlo como intercambio de datos entre servicios. Antes de popularizarse este formato, el más común era **XML (eXtended Marked language)** pero insertaba demasiadas etiquetas HTML (o XML) lo que lo hacía menos legible y más

complicado de decodificar. JSON por su parte al ser en sí un objeto JavaScript es ideal para aplicaciones que manejen este lenguaje.

Los detalles de implementación son que las propiedades del objeto deben ser Strings para que no haya problemas al codificarlo y decodificarlo.

Debemos tener en cuenta que algunos tipos de datos no se van a serializar igual. Por ejemplo los tipos **Nan** e **Infinity** se codifican como **null**. Y los objetos de tipo **Date** muestran la fecha en formato **ISO** y al reconstruirlo será un **String** sin poder acceder a los métodos que hereda de la clase Date . Tampoco se pueden serializar **funciones, expresiones regulares, errores y valores undefined**. El resto de primitivas y clases como **objects , arrays , strings , números, true , false y null** se pueden serializar.

Ventajas de este formato de datos [frente a XML](#):

- Más ligero (su estructura necesita menos elementos que XML) por lo que es ideal para peticiones AJAX.
- Más fácil de transformar a objeto Javascript (con eval se haría directo).

Particularidades del [formato JSON](#) frente a la notación literal de Javascript:

- Los pares nombre-valor van siempre con comillas dobles.
- JSON puede representar 6 tipos de valores: objetos, arrays, números, cadenas, booleanos y null.
- Las fechas no se reconocen como tipo de dato.
- Los números no pueden ir precedidos de 0 (salvo los decimales)

Las cadenas JSON deben ser convertidas a objetos Javascript para poder utilizarlas (y viceversa). Para ello podemos utilizar:

- [eval\(\)](#): No se recomienda utilizarlo directamente.
- [JSON.parse](#): Convierte una cadena JSON en un objeto Javascript hace eval pero comprueba el formato antes de hacerlo.
- [JSON.stringify](#): Convierte un objeto Javascript en una cadena JSON.
- [jQuery.parseJSON](#): con jQuery también podemos hacer el parseo del JSON.

El objeto JSON está disponible de forma nativa en los navegadores compatibles con ECMAScript 5

Ejemplo:

```
> JSON.parse('{"bar":"new property","baz":3}')
Object { bar="new property", baz=3}
```

```
> JSON.stringify({ breed: 'Turtle', occupation: 'Ninja' });
```

```
"{"breed": "Turtle", "occupation": "Ninja"}"
```

Ejemplo:

```
var usuario = {  
    "id": "012345888",  
    "username": "carlosrivera",  
    "password": "fkldfn4r09330adafnanf9843fbcdkjdkks",  
    "data": {  
        "name": "Carlos Rivera",  
        "email": "example@gmail.com",  
        "city": "Tucumán",  
        "country": "ARG"  
    },  
    "preferences": {  
        "contact": {  
            "email": true,  
            "notify": true  
        },  
        "interests": [  
            "javascript",  
            "html",  
            "css"  
        ]  
    }  
};
```

Si queremos acceder a una propiedad determinada podemos hacerlo como cualquier objeto:

```
usuario.data.city; // "Tucumán"
```

JSON.stringify

Si queremos serializarlo para poder realizar un intercambio de datos, debemos usar la función **JSON.stringify** que devuelve en un String la información del objeto que se le pasa por parámetro.

```
var jsonSerializado = JSON.stringify(usuario);
```

```
/* Devuelve:  
{ "id": "012345888", "username": "carlosrivera", "password": "fkldfn4r09330ada  
fnanf9843fbcdkjdkks", "data": { "name": "Carlos  
Rivera", "email": "example@gmail.com", "city": "Tucumán", "country": "ARG" }, "p  
references": { "contact": { "email": true, "notify": true } }, "interests": [ "javasc  
ript", "html", "css" ] } }"  
*/
```

Si ahora queremos acceder a las propiedades no podemos, porque se trata de un string.

```
jsonSerializado.data.city;  
/*  
Uncaught TypeError: Cannot read property 'city' of undefined  
at <anonymous>:2:21  
at Object.InjectedScript._evaluateOn (<anonymous>:895:140)  
at Object.InjectedScript._evaluateAndWrap (<anonymous>:828:34)  
at Object.InjectedScript.evaluate (<anonymous>:694:21)  
*/
```

JSON.parse

Para poder reconstruirlo a partir del string, tenemos la función **JSON.parse** que devuelve un objeto a partir del string que se pasa como parámetro. Tiene que estar correctamente formado, si no el método parse nos devolverá error.

```
var jsonReconstruido = JSON.parse(jsonSerializado);  
/*  
Object {id: "012345888", username: "carlosrivera", password:  
"fkldfn4r09330adafnanf9843fbcdkjdkks", data: Object, preferences:  
Object}  
*/
```

Ahora podemos acceder a sus propiedades como antes.

```
jsonReconstruido.data.city; // "Tucumán"
```

Más adelante veremos dónde se utilizan este tipo de datos, normalmente en intercambios de datos desde un servidor con llamadas HTTP, AJAX, o entre funciones

dentro de un mismo programa.

Web Storage

Web Storage es una característica de HTML5 que nos permite almacenar datos en el navegador del usuario.

Hay dos tipos de almacenamientos web que podemos usar:

Local Storage

Guarda información que permanecerá almacenada por tiempo indefinido; sin importar que el navegador se cierre.

Session Storage

Almacena los datos de una sesión y éstos se eliminan cuando el navegador se cierra.

Características de Local Storage y Session Storage:

- Permiten almacenar entre 5MB y 10MB de información; incluyendo texto y multimedia.
- La información está almacenada en la computadora del cliente y NO es enviada en cada petición del servidor, a diferencia de las cookies.
- Utilizan un número mínimo de peticiones al servidor para reducir el tráfico de la red.
- Previenen pérdidas de información cuando se desconecta de la red.
- La información es guardada por dominio web (incluye todas las páginas del dominio).

Local Storage

La mayoría de las **aplicaciones web**, **PWA** o incluso **aplicaciones móviles híbridas** suelen incluir formularios para que el usuario lo complete, ya sea con su perfil, para registrarse a una determinada plataforma, o para realizar una encuesta, entre muchos otros casos.

Pero, ¿qué pasa si durante el proceso de llenado del formulario, la aplicación web, browser o WebView se bloquea? La respuesta es evidente, los datos completados hasta ese momento se perderán.

Para prevenir la pérdida parcial de la información ingresada, e incluso para re-aprovechar la misma, podemos utilizar localStorage en el desarrollo de nuestro proyecto web o móvil, lo cual nos permitirá evitar la dependencia del sistema de Autocompletado, incluido en los navegadores web.

¿Qué es localstorage?

Cuando HTML5 comenzó a tomar gran protagonismo en el ecosistema web, trajo consigo

una serie de novedades beneficiosas; entre ellas localStorage. Esta propiedad, contenida usualmente en el objeto **window** de javascript, está presente tanto en los navegadores web de escritorio, móviles, como también en la aplicación independiente WebView.

A través de sus diversos métodos, localStorage nos permite manipular y almacenar localmente (o sea, del lado del usuario), información determinada en **formato clave, valor**.

Veamos entonces a localStorage como un almacén local de datos o más bien como una micro base de datos. Que sea local significa que la información será almacenada en el navegador del usuario. Dicha información quedará resguardada de forma permanente hasta que el usuario manualmente la elimina, por ejemplo limpiando la cache y datos del navegador o a través de alguna función programática.

Veamos a continuación en una tabla los métodos y propiedades que trae el objeto localStorage:

Método	Descripción
clear()	Elimina toda la información almacenada.
setItem(clave, valor)	Guarda una clave determinada, y un valor asignado a ésta.
getItem(clave)	Obtiene un dato determinado. Esta función requiere pasar como parámetro la clave de la cual deseamos obtener su valor asignado.
length	Nos permite conocer el total de claves almacenadas por nuestra aplicación.

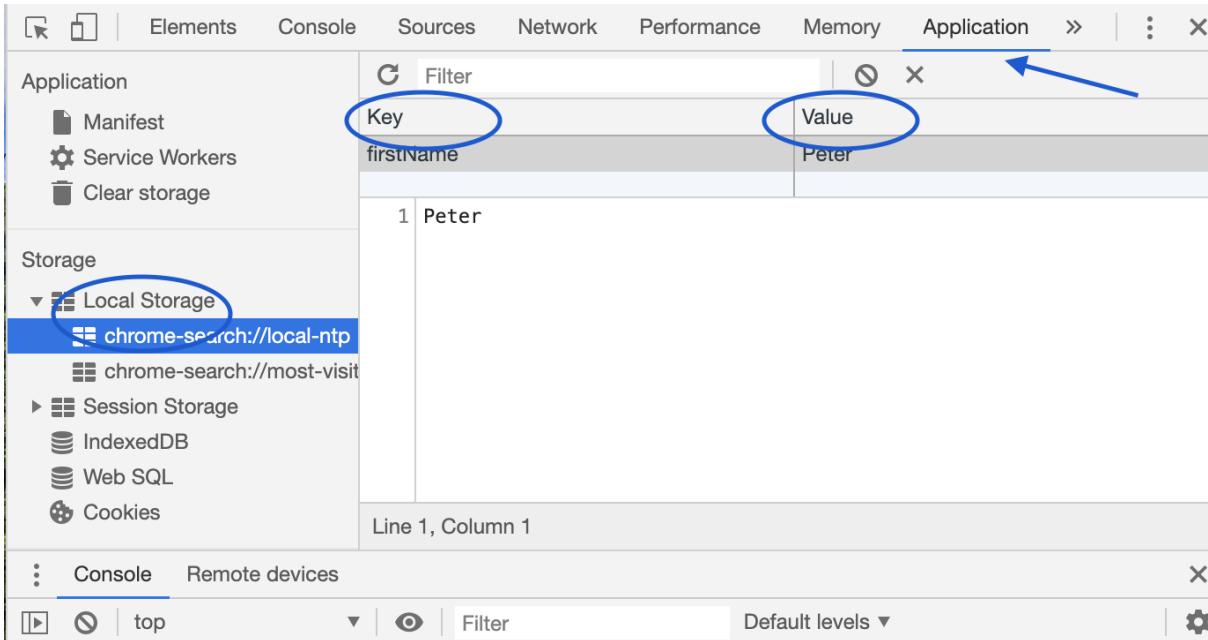
Veamos un ejemplo:

```
// verificar si el objeto localStorage existe
if(localStorage) {
    // almacenar datos
    localStorage.setItem("firstName", "Peter");
    // recuperar el dato guardado
    alert("Hola, " + localStorage.getItem("firstName"));
} else {
    alert("Tu navegador no soporta local storage :(");
}
```

En el ejemplo anterior verificamos que el objeto localStorage existe (en la mayoría de los

navegadores modernos el objeto localStorage existe). Luego con el método **setItem** guardamos en la clave **fisrtName** el valor **Peter**, en la siguiente línea con un **alert** y usando el método **getItem** con la clave **firstName**, recuperamos el valor guardado. Si el navegador no soporta localStorage mostramos una alerta.

Cuando ejecutamos el código anterior en el navegador, en Google Chrome podemos ver la información almacenada en el local storage a través del menú de **Developer Tools** o **Herramientas del desarrollador**, en la sección de Application como se muestra en la siguiente imagen.



The screenshot shows the Google Chrome Developer Tools interface with the Application tab selected. In the left sidebar under Storage, Local Storage is expanded, showing items like chrome-search://local-ntp, chrome-search://most-visit, Session Storage, IndexedDB, Web SQL, and Cookies. One item, chrome-search://local-ntp, is highlighted with a blue oval. In the main content area, there's a table with two columns: Key and Value. The Key column contains 'firstName' and '1'. The Value column contains 'Peter'. A blue oval highlights the 'Value' column header. Below the table, the text 'Line 1, Column 1' is visible. At the bottom of the tools, a code editor window displays the following JavaScript code:

```

> // verificar si el objeto localStorage existe
if(localStorage) {
  // almacenar datos
  localStorage.setItem("firstName", "Peter");
  // recuperar el dato guardado
  alert("Hola, " + localStorage.getItem("firstName"));
} else {
  alert("Tu navegador no soporta local storage :(");
}
< undefined
  
```

Veamos otros ejemplo. Vamos a crear un formulario con 2 campos donde el usuario podra cargar su nombre y apellido y guardarlos posteriormente en el local storage.

Parte html:

```

<html>
<body>
  <form>
    <input type="text" name="firstName">
  
```

```

<input type="text" name="lastName">
<button type="submit" id="save">Guardar</button>
<button id="btnRemoveData">Eliminar</button>
</form>
<div id="dataContent"></div>
</body>
</html>

```

Parte Javascript:

```

function ready() {
  //console.log('DOM is ready');
  let firstName = localStorage.getItem('firstName');
  let lastName = localStorage.getItem('lastName');
  //Creamos elementos en el dom
  let elem1 = document.createElement('p');
  elem1.innerHTML = "Nombre: " + firstName;
  let elem2 = document.createElement('p');
  elem2.innerHTML = "Apellido: " + lastName;
  //Agregamos los elementos creados al div con id dataContent
  let dataContent = document.getElementById('dataContent');
  dataContent.appendChild(elem1);
  dataContent = document.getElementById('dataContent');
  dataContent.appendChild(elem2);
}

function saveData(e){
  e.preventDefault();
  e.stopPropagation();
  //Get data from dom
  let firstName = document.forms[0].elements[0].value;
  let lastName = document.forms[0].elements[1].value;
  //Save data
  localStorage.setItem('firstName', firstName);
  localStorage.setItem('lastName', lastName);
}

function removeData(e){
  //Eliminamos los datos del local storage
  localStorage.clear('firstName');
  localStorage.clear('lastName');
  //Eliminamos el nodo dataContent
}

```

```
let dataContent = document.getElementById('dataContent');
dataContent.parentNode.removeChild(dataContent);
}

//agregamos un manejador de eventos para cuando la página este lista
document.addEventListener("DOMContentLoaded", ready);
//agregamos un manejador de eventos para capturar los datos
//del formulario cuando el usuario presiona el botón Guardar
document.forms[0].addEventListener("submit", saveData, false);
//agregamos un manejador de eventos cuando el usuario
//presiona el botón Eliminar
let btnRemoveData = document.getElementById('btnRemoveData');
btnRemoveData.addEventListener('click', removeData, false);
```

Resultado:



A screenshot of a web application interface. It features two input fields side-by-side, followed by a 'Guardar' button. Below these elements is a 'Eliminar' button.

Nombre: juan

Apellido: perez

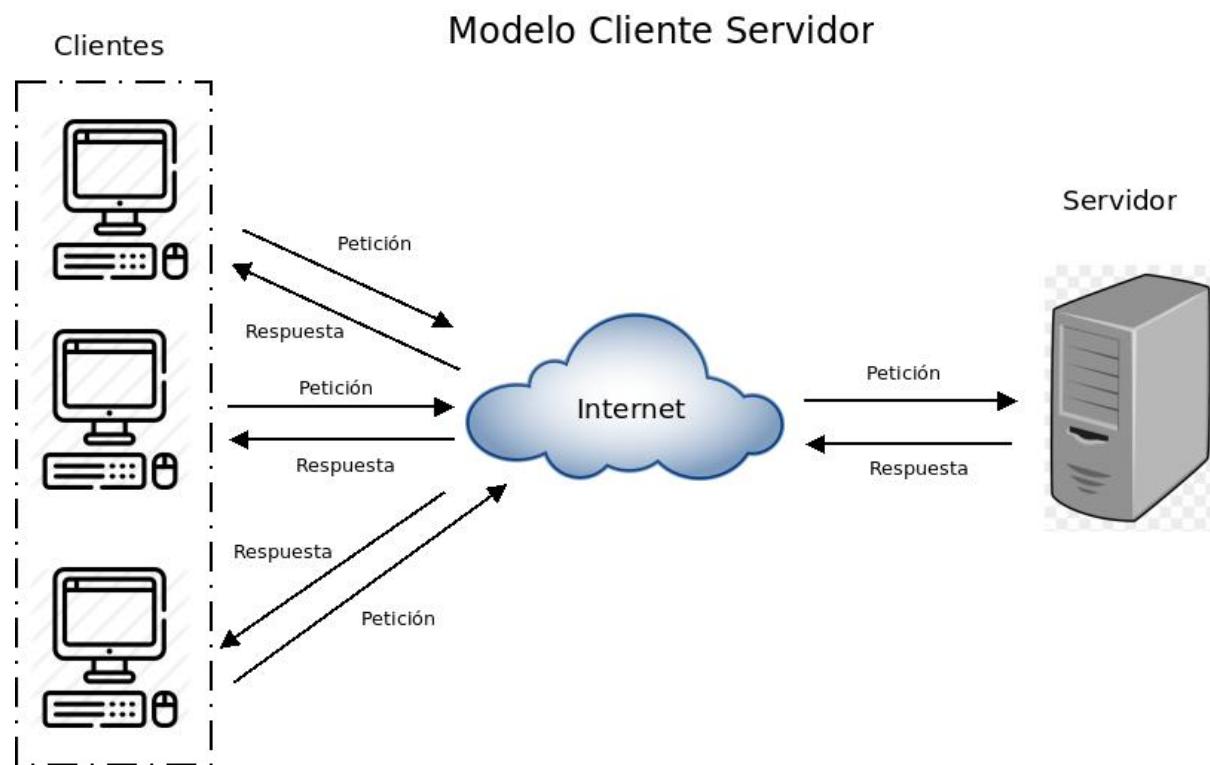
El ejemplo anterior básicamente carga los datos almacenados en local storage cuando la página carga el DOM complemento. Esto es supervisado a través del evento **DOMContentLoaded**. Cuando el usuario hace click en el botón de **Guardar**, se activa la función **saveData()** adjunta al manejador del evento **submit**. Esta almacena la información en el local storage. Finalmente cuando el usuario presiona el botón **Eliminar**, se activa la función manejadora del evento click asociado a este botón que se denomina **removeData()**, la cual simplemente elimina los datos del local storage y elimina el nodo **dataContent** que es donde se visualiza la información cargada.

Ajax

Antes de comenzar con el concepto de Ajax, vamos a hablar sobre el concepto de arquitectura cliente servidor, que es el más común utilizado en la web. Este concepto nos permitirá entender cómo es que funcionan las páginas en internet, desde el momento en que un usuario solicita una escribiendo en la barra de direcciones del navegador web.

Arquitectura Cliente Servidor

Este modelo es uno de los principales usados en muchísimos servicios y protocolos de Internet, por lo que para todos aquellos que quieren aprender más sobre la web y cómo funciona, entender el concepto de modelo cliente servidor se vuelve algo indispensable.



Arquitectura Cliente Servidor

La arquitectura cliente servidor tiene dos partes claramente diferenciadas, por un lado la parte del **servidor** y por otro la parte de **cliente** o grupo de clientes donde lo habitual es que un servidor sea una máquina bastante potente con un hardware y software específico que actúa de depósito de datos y funcione como un sistema gestor de base de datos o aplicaciones web.

En esta arquitectura el cliente puede ser un teléfono, tablet, una pc, etc, cualquier dispositivo que sea capaz de conectarse a un red como internet; que solicitan varios servicios al servidor, mientras que un servidor es una máquina que actúa como depósito de datos y funciona como un sistema gestor de base de datos, este se encarga de dar la respuesta demandada por el cliente.

El más claro ejemplo de uso de una arquitectura cliente servidor es la **red de Internet** donde existen ordenadores de diferentes personas conectadas alrededor del mundo, las cuales se conectan a través de los servidores de su proveedor de Internet por ISP donde son dirigidos a los servidores de las páginas que desean visualizar y de esta manera la

información de los servicios requeridos viajan a través de Internet dando respuesta a la solicitud demandada.

La principal importancia de este modelo es que permite conectar a varios clientes a los servicios que provee un servidor y como sabemos hoy en día, la mayoría de las aplicaciones y servicios tienen como gran necesidad que puedan ser consumidos por varios usuarios de forma simultánea.

Componentes cliente servidor

Para entender este modelo vamos a nombrar y definir a continuación algunos conceptos básicos que lo conforman.

- **Red:** Una red es un conjunto de clientes, servidores y base de datos unidos de una manera física o no física en el que existen protocolos de transmisión de información establecidos como por ejemplo el protocolo http que se usa para la transmisión de páginas web.
- **Cliente:** El concepto de cliente hace referencia a un demandante de servicios, este cliente puede ser un ordenador como también una aplicación de informática (Ej. el navegador de internet), la cual requiere información proveniente de la red para funcionar.
- **Servidor:** Un servidor hace referencia a un proveedor de servicios, este servidor a su vez puede ser un ordenador o una aplicación informática, la cual envía información a los demás agentes de la red. Ejemplos de servidores pueden ser:
 - Servidor de páginas web: Apache Server, Nginx, IIS.
 - Servidores de correo electrónico
 - Servidores de almacenamiento de archivos
- **Protocolo:** Un protocolo es un conjunto de normas o reglas establecidos de manera clara y concreta sobre el flujo de información en una red estructurada. Ej el protocolo http que es el que se utiliza para la web.
- **Servicios:** Un servicio es un conjunto de información que busca responder las necesidades de un cliente, donde esta información pueden ser una página web, mail, música, mensajes simples entre software, videos, etc.
- **Base de datos:** Son bancos de información ordenada, categorizada y clasificada que forman parte de la red, que son sitios de almacenaje para la utilización de los servidores y también directamente de los clientes. Ej Mysql, MongoDB, etc.

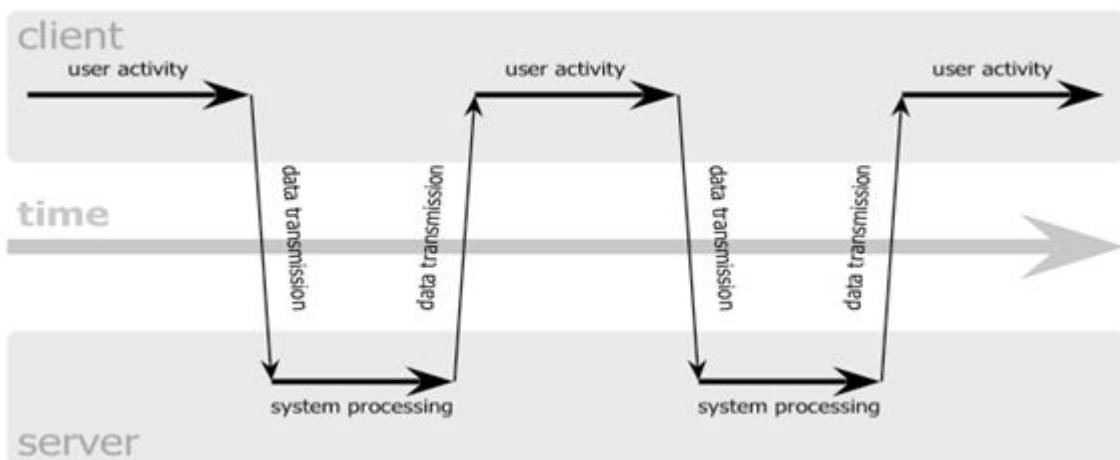
Ajax introducción

AJAX es el acrónimo de **Asynchronous JavaScript And XML**. Es el uso de JavaScript para realizar llamadas o **peticiones asíncronas** al servidor utilizando XML. La parte de XML ha sido sustituida hoy en día por JSON, que es un formato más amigable para el intercambio de datos, aunque se sigue utilizando la X en el acrónimo AJAX para expresar esta tecnología.

JavaScript se hizo muy popular por la llegada de esta tecnología y prueba de ello son las aplicaciones que surgieron y siguen utilizándose hoy en día como Gmail, Facebook, Twitter, etc...

Ajax agrupa un conjunto de tecnologías cuyo eje central son las peticiones asíncronas al servidor a través del objeto [XMLHttpRequest\(\) \(XHR\)](#). La aplicación de esta técnica dio lugar a las llamadas aplicaciones AJAX, donde *no es necesario refrescar la página* para obtener nuevo contenido.

Modelo de aplicación web clásico (Síncrono)

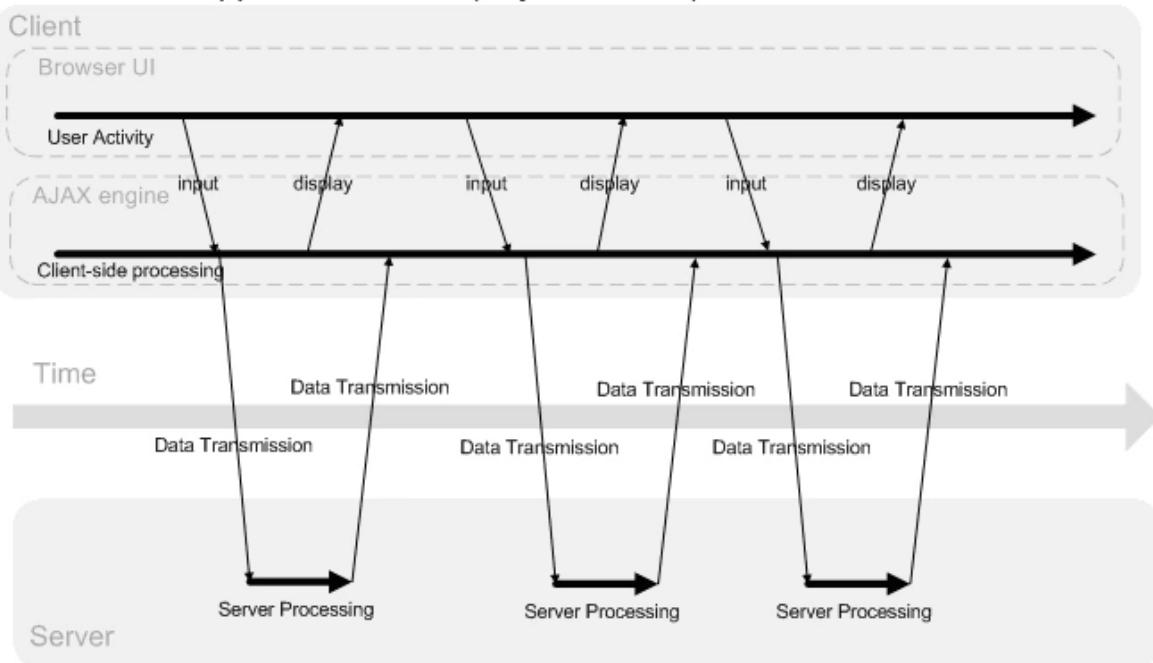


En un modelo de programación síncrona, las cosas suceden una a la vez. Cuando se llama a una función que realiza una acción de larga duración, sólo vuelve cuando la acción ha finalizado y puede devolver el resultado. Esto detiene el programa durante el tiempo que dure la acción, lo cual significa que el usuario no puede realizar ninguna otra acción hasta que la operación no haya finalizado. En internet cuando un usuario ingresa a una página el mismo tiene que esperar a que la misma se cargue (html, archivos css, archivos js, imágenes, etc) complementariamente para poder utilizarla, y si navega a otra página de nuevo tiene que esperar a que se cargue completamente y así sucesivamente. Este modelo síncrono ha existido desde que nació la web en los 90 básicamente. El problema es que las páginas web cada vez son más pesadas y ese tiempo de carga sino se controla puede ser muy alto lo que puede causar una mala experiencia de usuario.

Modelo de aplicación web con ajax (Asíncrono)

Un modelo asíncrono permite que sucedan múltiples cosas al mismo tiempo. Al iniciar una acción, el programa continúa ejecutándose. Cuando la acción termina, el programa es informado y tiene acceso al resultado (por ejemplo, los datos leídos desde el disco). En una página web significa que una vez la misma se carga completamente al iniciar por primera vez, el usuario al navegar por las distintas secciones de la página ya no tendría que recargar toda la página nuevamente, sino sólo las secciones que correspondan.

AJAX Web Application Model (Asynchronous)



Qué significa AJAX

- **Asynchronous** porque después de hacer una petición HTTP, el usuario no necesita esperar a una respuesta, sino que puede seguir haciendo otras cosas y ser notificado cuando llegue la respuesta.
- **JavaScript** porque creamos los objetos XHR con Javascript.
- **XML** porque inicialmente era el formato standard de intercambio de datos. Actualmente una petición HTTP suele devolver JSON (o HTML)

La mayor limitación de una petición AJAX es que no puede acceder a datos de un dominio diferente al que estamos navegando (cross-domain).

Pero hay maneras de solucionar este problema: [JSONP](#) y [CORS](#)

Iniciando una petición o request AJAX

Una petición es un recurso(página web, imagen, un archivo, etc) que el cliente solicita al servidor. Desde que el usuario presiona enter al ingresar una url en el browser, estamos realizando una petición solicitando una página web a un servidor que se encuentra en alguna parte del planeta. El servidor nos retorna el archivo html que luego el navegador interpreta y dibuja los elementos en pantalla. Ahora veremos como se hace un request usando Ajax.

Para iniciar una petición ajax usamos el objeto de javascript [XMLHttpRequest](#).

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = myCallback;
xhr.open('GET', url, true);
xhr.send();
```

Las peticiones AJAX las hacemos a través de objetos XHR.

Para crear un objeto XHR utilizamos el objeto **XMLHttpRequest()** que es nativo en IE7+, Safari, Opera y Firefox.

```
var xhr = new XMLHttpRequest();
```

Una vez creado el objeto XHR, capturamos el evento **onreadystatechange** de este objeto y le pasamos una función manejadora del evento.

```
xhr.onreadystatechange = myCallback;
```

Después hay que llamar al método **open()** pasandole los parametros de la petición.

```
xhr.open('GET', 'https://google.com', true);
```

- El 1er parámetro es el tipo de petición (GET, POST,...).
- El 2º parámetro es la URL. En este ejemplo intentamos obtener la página de google
- El 3er parámetro indica si la petición es asíncrona (true) o síncrona (false)

Después hay que llamar al método **send()** para hacer la petición.

```
xhr.send();
```

Procesando la respuesta

```
function myCallback() {
  if (xhr.readyState < 4) {
    return; // not ready yet
  }
  if (xhr.status !== 200) {
    alert('Error!'); // the HTTP status code is not OK
    return;
}
```

```
// all is fine, do the work
alert(xhr.responseText);
}
```

El objeto XHR tiene una propiedad llamada **readyState** y cada vez que cambia esta propiedad se dispara el evento **onreadystatechange**.

Los posibles valores de readyState son:

- 0 : UNSENT (No se ha llamado al método open())
- 1 : OPENED (Se ha llamado al método open())
- 2 : HEADERS_RECEIVED (Se ha llamado al método send())
- 3 : LOADING (Se está recibiendo la respuesta)
- 4 : DONE (**se ha completado la operación**)

Cuando **readyState** llega a **4** significa que ya se ha recibido una respuesta

Una vez hemos recibido la respuesta hay que chequear el estado de la misma en la propiedad **status** del objeto XHR

El valor que nos interesa para esta propiedad es **200 (OK)**.

Ejemplo completo

Veamos un ejemplo de cómo utilizar llamadas asíncronas en una página web, imaginemos que tenemos un servidor o nos proporcionan un servicio que a través de una URL nos devuelve una cantidad de datos en formato JSON. por ejemplo la siguiente dirección: <http://jsonplaceholder.typicode.com/photos> , que devuelve lo siguiente:

```
[
  {
    "albumId": 1,
    "id": 1,
    "title": "accusamus beatae ad facilis cum similique qui sunt",
    "url": "http://placehold.it/600/92c952",
    "thumbnailUrl": "http://placehold.it/150/30ac17"
  },
  {
    "albumId": 1,
    "id": 2,
    "title": "reprehenderit est deserunt velit ipsam",
    "url": "http://placehold.it/600/771796",
    "thumbnailUrl": "http://placehold.it/150/dff9f6"
  },
]
```

```
{  
  "albumId": 1,  
  "id": 3,  
  "title": "officia porro iure quia iusto qui ipsa ut modi",  
  "url": "http://placehold.it/600/24f355",  
  "thumbnailUrl": "http://placehold.it/150/1941e9"  
}  
]
```

¿Cómo hacemos para llamar a esa URL dentro de nuestra página web de manera asíncrona (sin recargar la página)?

```
// Creamos el objeto XMLHttpRequest  
var xhr = new XMLHttpRequest();  
  
// Definimos la función que manejará la respuesta  
function reqHandler () {  
  if (this.readyState === 4 && this.status === 200) {  
    /* Comprobamos que el estado es 4 (operación completada)  
     * Los estados que podemos comprobar son:  
     * 0 = UNSEND (No se ha llamado al método open())  
     * 1 = OPENED (Se ha llamado al método open())  
     * 2 = HEADERS_RECEIVED (Se ha llamado al método send())  
     * 3 = LOADING (Se está recibiendo la respuesta)  
     * 4 = DONE (se ha completado la operación)  
     * y el código 200 es el correspondiente al OK de HTTP de  
     * que todo ha salido correcto.  
    */  
    console.log(this.responseText);  
  }  
}  
  
// Asociamos la función manejadora al evento onreadystatechange  
xhr.onreadystatechange = reqHandler;  
// Abrimos la conexión hacia la URL, indicando el método HTTP, en este  
// caso será GET  
xhr.open('GET', 'http://jsonplaceholder.typicode.com/photos', true);  
// Enviamos la petición  
xhr.send();
```

Esto es lo que mostrará en la consola:

```
/*
{
  "albumId": 1,
  "id": 1,
  "title": "accusamus beatae ad facilis cum similique qui sunt",
  "url": "http://placehold.it/600/92c952",
  "thumbnailUrl": "http://placehold.it/150/30ac17"
},
{
  "albumId": 1,
  "id": 2,
  "title": "reprehenderit est deserunt velit ipsam",
  "url": "http://placehold.it/600/771796",
  "thumbnailUrl": "http://placehold.it/150/dff9f6"
},
...
]
*/
```

Como es un objeto JSON, podemos acceder a sus propiedades, iterarlo, etc.. Entonces imaginemos que el anterior código JavaScript lo tenemos en una página HTML que contiene un **<div id='respuesta'></div>**:

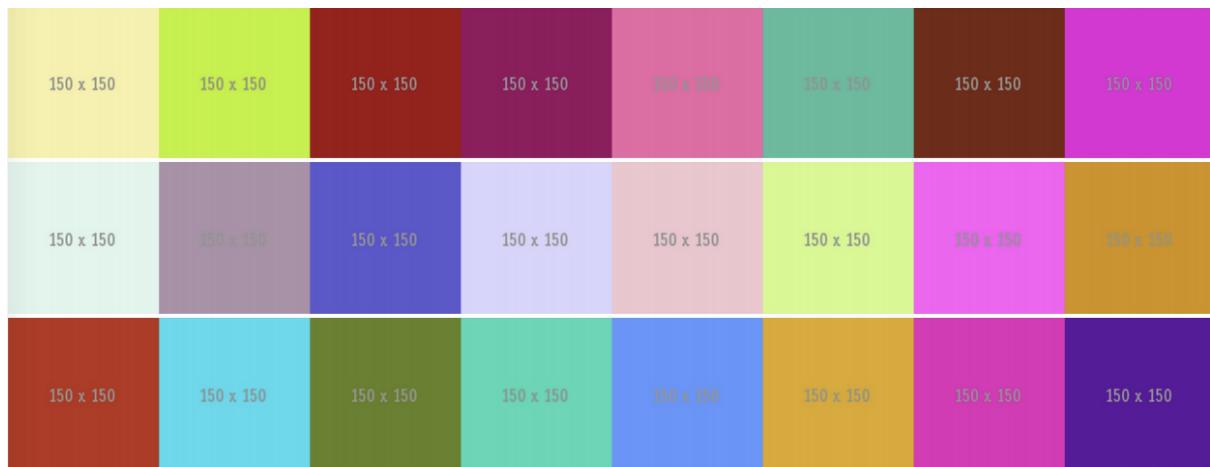
```
<!-- Página HTML -->
<html>
  <body>
    <div id="respuesta"></div>
  </body>
</html>
```

si la función **reqHandler** la modificamos de esta manera:

```
function reqHandler () {
  if (this.readyState === 4 && this.status === 200) {
    var respuesta = JSON.parse(this.responseText);
    var respuestaHTML = document.querySelector('#respuesta');
    var tpl = '';
    respuesta.forEach(function (elem) {
      tpl += '<a href="' + elem.url + '">' +
        '',
        '</a>',
    });
    respuestaHTML.innerHTML = tpl;
  }
}
```

```
+ '<br/>',
+ '<span>' + elem.title + '</span>';
});
respuestaHTML.innerHTML = tpl;
}
}
```

Obtenemos como resultado lo siguiente:



Más adelante cuando veamos las nuevas características de ES6 veremos un método más simple para hacer peticiones http.