JavaScript Para Gatos

## Una introducción para nuevos programadorescat

### Tan simple que hasta tu amigo humano puede también hacerlo!

JavaScript es un lenguaje de programación o, en otras palabras, un medio por el cual la computadora es instruída para hacer cosas. Así como cualquiera de nosotros controlamos a los humanos con silbidos y maullidos, uno puede controlar computadoras con enunciados escritos en un lenguaje de programación. Todos los navegadores web entienden JavaScript y tu puedes tomar ventaja de eso para hacer que las páginas web hagan cosas locas!

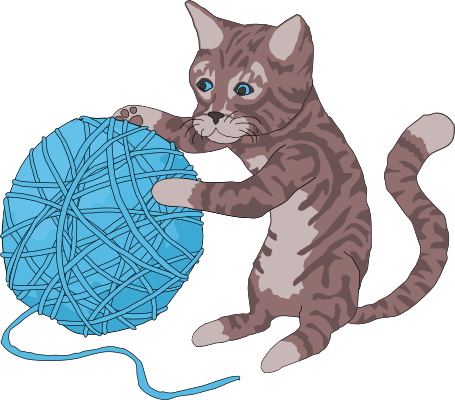
JavaScript empezó como una manera de hacer páginas web más interactivas. Hoy en día JavaScript corre no solo en navegadores — también corre en servidores web, teléfonos y hasta robots! Esta página te va a enseñar una base de JavaScript para que puedas arrancar instantáneamente\*.

\* Tiempo real del tutorial: Más que instantáneo. Probablemente una hora o dos. Además como sos un gato te debe gustar más estar tirado al sol que corriendo.

## Tabla de contenidos

* [La base](https://jsparagatos.com/" \l "basics)
* [Strings](https://jsparagatos.com/" \l "strings)
* [Valores y variables](https://jsparagatos.com/" \l "values)
* [Usando funciones](https://jsparagatos.com/" \l "functions)
* [Funciones del lenguaje](https://jsparagatos.com/" \l "standard-library)
* [Código de terceros](https://jsparagatos.com/" \l "third-party-javascript)
* [Creando nuevas funciones](https://jsparagatos.com/" \l "writing-functions)
* [Ciclos](https://jsparagatos.com/" \l "loops)
* [Arrays](https://jsparagatos.com/" \l "arrays)
* [Objetos](https://jsparagatos.com/" \l "objects)
* [Callbacks](https://jsparagatos.com/" \l "callbacks)
* [Lectura recomendada](https://jsparagatos.com/" \l "recommended-reading)

## No seas un gato miedoso



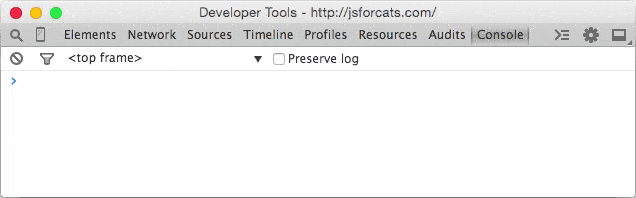
Siempre vas a caer de pie — incluso programando! A diferencia de [**patear un vaso de agua**](https://jsparagatos.com/images/dealwithit.gif) sobre tu laptop, no hay nada en estos tutoriales que vaya a dañar tu computadora de ninguna manera, incluso si escribes mal un comando o haces click en un botón equivocado. Como los gatos, los programadores cometen errores todo el tiempo: errores de escritura, olvido de cierre de comillas o llaves, y ser olvidadizos acerca de como funcionan algunas funciones básicas. Los programadores se preocupan más por hacer las cosas funcionar eventualmente en vez de tratar de hacerlas funcionar la primera vez que se escriben. La mejor forma de aprender es cometiendo errores!

Es por eso que no tienes que ser un gato miedoso! La peor cosa que puede pasar es que tengas que refrescar esta página en tu navegador web si estás atascado. Tampoco te preocupes, esto pasa muy rara vez.

## [#](https://jsparagatos.com/" \l "basics) La base

Hay JavaScript corriendo en esta página en este momento. Juguemos un poco con eso. Por simplicidad voy a asumir que estás usando Google Chrome para leer esta página (si no es así va a ser probablemente más fácil para los dos si sigues el tutorial usando Chrome).

Primero, haz click derecho en cualquier parte de la pantalla y selecciona Inspeccionar elemento (o Inspect element en inglés), después click en el tab que dice Console. Deberías ver una cosa que se parece a esto:



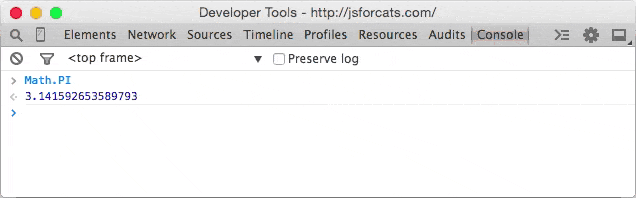
Esta es una consola, también llamada "línea de comando" o "terminal". Básicamente es una forma de escribir una cosa a la vez en la computadora e inmediatamente obtener la respuesta. Existen herramientas de aprendizaje súper útiles (Yo todavía uso la consola casi todos los días que escribo código).

La consola hace algunas cosas realmente geniales. Comencé escribiendo algo y la consola está ayudandome dandome una lista de todas las posibles cosas que puedo continuar escribiendo! Otra cosa que puedes hacer es escribir 1 + 1 en la consola y después apretar Enter y ver que pasa.

Usar la consola es una parte importante a la hora de aprender JavaScript. Si no sabes como funciona algo o que comando sirve para qué, puedes ir a la consola y averiguarlo! El siguiente es un ejemplo:

### [#](https://jsparagatos.com/" \l "strings) Strings

Como soy un gato quiero reemplazar cada instancia de la palabra perro en la Internet con esos malditos perros. Primero voy a la consola y escribo unas pocas sentencias que contienen la palabra perro al menos una vez. En JavaScript un conjunto de letras, números, palabras o cualquier otra cosa es conocido como un String (o cadena de caracteres). Los strings deben empezar y terminar con una comilla. Da igual usar comilla simple ' o doble ", siempre y cuando uses el mismo símbolo al principio y al final.



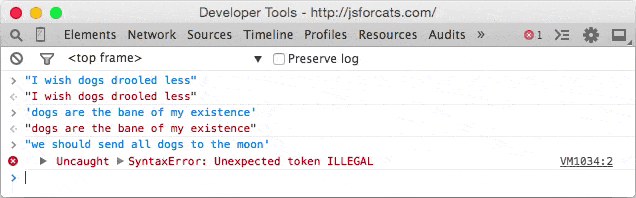
Ves el horrible mensaje de error? No te preocupes - no rompiste ninguna ley. Syntax Error ILLEGAL es como suena cuando un robot te quiere decir que tu programa tiene un problema. Las primeras dos sentencias coinciden con el tipo de comilla usado al comienzo y final de la sentencia pero cuando los mezclamos la computadora enloquece, es como mezclar sandía con vino.

Ok, para arreglar una de esas sentencias (reemplazando perro por nuestra versión mejorada) tenemos que primero guardar la sentencia original para poder llamarla de nuevo cuando hagamos nuestra magia de reemplazo. Ves como el string se repite en rojo cuando escribo en la consola? Esto es porque no dijimos que se guarde la sentencia en ningún lado y por eso no las devuelve (O nos da un error cuando cometemos algún error).

### [#](https://jsparagatos.com/" \l "values) Valores y variables

Los valores son los componentes más simples en JavaScript. 1 es un valor, true (que representa un valor lógico de verdad) es un valor, "hola" es un valor, function() {} es un valor, la lista sigue! Existen muchos tipos de valores en JavaScript pero no necesitamos conocerlos todos ahora — los vas a aprender naturalmente mientras más código escribas!

Para guardar valores usamos cosas llamadas variables. La palabra 'variable' significa 'puede cambiar' y es usado porque las variables pueden guardar distintos tipos de valores y cambiar su valor muchas veces. Las variables son bastante parecidas a un buzón. Ponemos algo en una variable, como una sentencia, y después le damos a la variable una dirección que puede usar para ir a buscar esa sentencia más tarde. En la vida real los buzones necesitan números de código postal pero en JavaScript usualmente se reemplazan con letras en minúscula o números sin espacios.



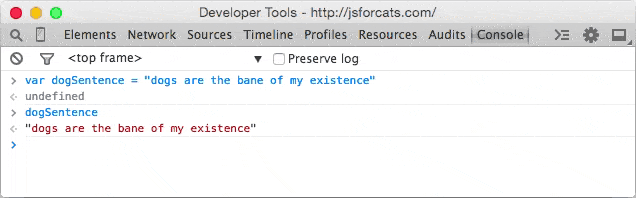
var es una abreviatura de variable y el símbolo = significa guarda la cosa del lado derecho en la cosa del lado izquierdo. Además como puedes ver, ahora que estamos guardando nuestra sentencia en una variable la consola no devuelve la sentencia, sino que nos da undefined como respuesta, lo que significa no hay nada para devolver.

Si simplemente escribes el nombre de una variable en la consola va a imprimir el valor guardado en esa variable. Una nota sobre las variables es que por defecto se pierden cuando cambias a una página diferente. Si fuera a refrescar la página en Chrome, por ejemplo, mi variable sentenciaPerro desaparecería para siempre, como si nunca hubiera existido. Pero no te preocupes por esto demasiado — puedes apretar las flechas para arriba y abajo en el teclado cuando estás en la consola y obtener las sentencias que fuiste escribiendo últimamente.

### [#](https://jsparagatos.com/" \l "functions) Usando funciones

Ahora que tenemos nuestra sentencia guardada en una variable, vamos a cambiar la palabra guardada en ella! Podemos hacer esto creando una función. Las funciones son un tipo de valor que realiza justamente una función específica (También conocido como propósito o acción) por nosotros. Llamarlas "acciones" suena extraño y supongo que por eso eligieron "función" en su lugar.

JavaScript tiene una función llamada replace (reemplazar en inglés) que hace exactamente lo que queremos! Las funciones toman valores en sus paréntesis (cero, uno o más) y devuelve o bien nada (undefined) o el string modificado. La función replace está disponible para usar sobre cualquier string y toma 2 valores: los caracteres que queremos sacar y los caracteres que los van a reemplazar. Se torna un poco confuso describir estas cosas así que acá tenemos un ejemplo visual:



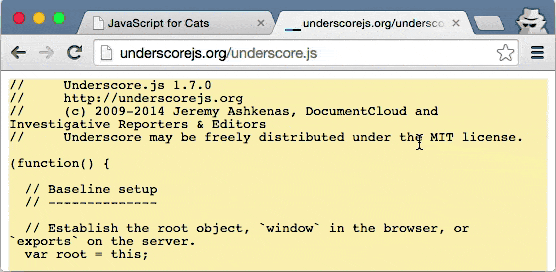
Notas como el valor de sentenciaPerro es el mismo incluso después de correr replace sobre él? Esto se debe a que la función replace, (y la mayoría de funciones de JavaScript para estos propósitos) toman el valor que le damos y retornan un nuevo valor, sin modificar el valor que le pasamos. Como no queremos guardar el resultado (no hay = en el lado izquierdo de la operación de reemplazo) se imprime el valor retornado en la consola.

### [#](https://jsparagatos.com/" \l "standard-library) Funciones del lenguaje

Te debes estar preguntando que otras funciones están disponibles en JavaScript. La respuesta: UN MONTON. Hay muchas bibliotecas estándar integradas de las cuales puedes aprender en MDN (Un sitio de Mozilla que tiene mucha y muy buena información sobre tecnologías web). Por ejemplo [**acá está la página de MDN sobre las funciones relativas a la matemática de JavaScript**](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Math).

### [#](https://jsparagatos.com/" \l "third-party-javascript) Código de terceros

También existe un montón de código JavaScript disponible que no forma parte de la biblioteca estándar. Se suele referir al JavaScript escrito por tercero como una "biblioteca" o "plugin". Uno de mis favoritos se llama Underscore.js. Vayamos a buscarlo y cargarlo en nuestra página! Primero ve al sitio de Underscore, [**http://underscorejs.org/**](http://underscorejs.org/), haz click en el link de descarga (En general uso la versión de desarrollo porque son fáciles de leer pero cualquiera de las 2 te va a dar la misma funcionalidad), y copia todo el código en tu portapapeles (puedes usar Seleccionar todo haciendo click derecho). Después págalo en tu consola y apretá enter. Ahora tu navegador tiene una nueva variable: \_. Underscore te da una gran cantidad de funciones útiles para jugar. Vamos a aprender a usarlas más adelante.



### [#](https://jsparagatos.com/" \l "writing-functions) Creando nuevas funciones

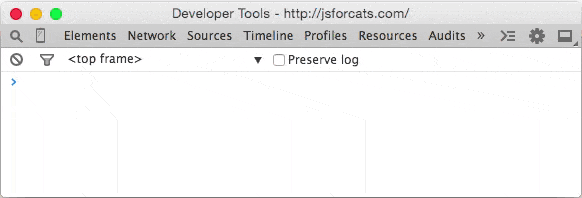
No estás limitado a usar funciones de otros — puedes también escribirlas vos. Es muy simple! Hagamos una función llamada ponerleOnda que agregue unos puntos de exclamación al final de un string.

function ponerleOnda(string) {

return string + '!!!!'

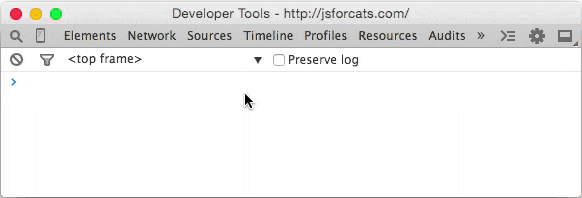
}

En mi cabeza lo leo en voz alta como: "hay una función llamada 'ponerle onda' que toma un string y devuelve una nueva copia del string que tiene algunos puntos de exclamación al final". Así es como escribo esto en la consola manualmente sin escribir la función:

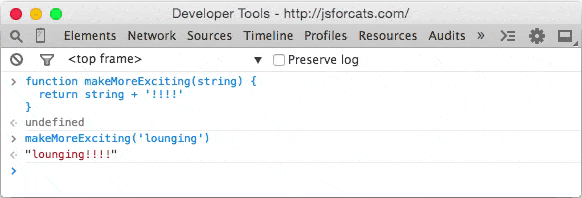


La expresión string + !!!! devuelve un nuevo string y nuestra variable llamada string queda igual que antes (ya que nunca la actualizamos usando =).

Usemos nuestra función en vez de hacerlo manualmente. Primero, pega la función en la consola y después llama a la función pasandole un string:



Puedes también llamar a la misma función pasandole una variable que apunta a un string (en el ejemplo de arriba escribimos el string directo como un valor en vez de guardarlo como una variable primero):



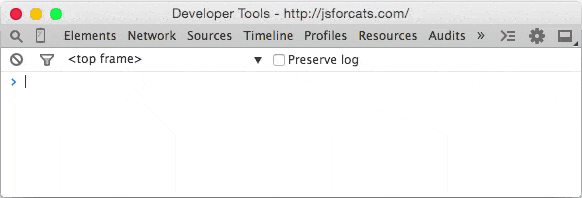
La linea ponerleOnda(sentencia) es equivalente a decir sentencia + '!!!!'. Que pasa si queremos modificar en el lugar (conocido como actualizar) el valor de sentencia? Simplemente guarda el valor de retorno de la función en la variable sentencia:

var sentencia = "hora de la siesta"

sentencia = ponerleOnda(sentencia)

Ahora sentencia va a tener los signos de exclamación al final! Date cuenta que vos solo usaste var cuando inicializaste la variable — la primera vez que la usaste. Después no deberías usar var a menos que quieras re-inicializar (resetear/limpiar/vaciar) la variable.

Que pasa si sacamos el return de la función?



Por qué sentencia está vacía? Porque las funciones devuelven undefined por defecto! puedes elegir devolver cualquier valor usando return. Las funciones deben tomar un valor y, si cambian el valor o crean uno nuevo que se va a usar más tarde, devuelven un valor (curiosidad: un término cool para este estilo es programación funcional). Acá tenemos otra función que no devuelve ningún valor pero en cambio usa un método diferente para mostrarnos el resultado:

function gritalo(string) {

string = string.toUpperCase()

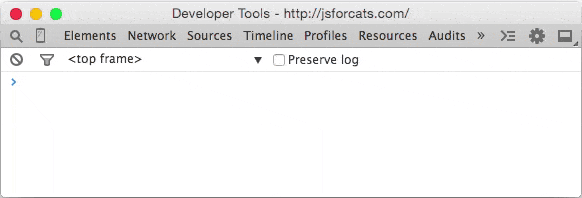
string = ponerleOnda(string)

console.log(string)

}

Esta función, gritalo, usa nuestra función ponerleOnda y el método nativo de String [**toUpperCase**](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/toUpperCase). Los métodos son solo un nombre para una función cuando pertenece a algo — en este caso toUpperCase es una función que pertenece a String entonces decimos que es un método o una función. ponerleOnda en cambio no pertenece a nadie entonces sería tecnicamente incorrecto referirse a ella como un método (es confuso, lo se).

La última linea de la función es otra función nativa que simplemente toma valores que le das y los imprime en la consola.



Entonces hay algo mal con esta función gritalo? Depende! Hay 2 tipos principales de funciones:

* Funciones que modifican o crean valores y los retornan
* Funciones que toman valores y hacen cosas con ellos que no se pueden retornar

console.log es un ejemplo del segundo grupo: Imprime cosas en tu consola — una acción que podes ver con tus propios ojos pero que no puede ser representada como un valor de JavaScript. Mi propia regla del pulgar es tratar de mantener los 2 tipos de funciones separadas, así que podemos reescribir la función gritalo de este modo:

function gritalo(string) {

string = string.toUpperCase()

return ponerleOnda(string)

}

console.log(gritalo("no le temo a los humanos"))

De este modo gritalo se vuelve más genérica, es decir que hace una o dos cosas simples y no sabe nada sobre imprimir a la consola — esa parte siempre puede ser programada después, afuera de la definición de la función.

### [#](https://jsparagatos.com/" \l "loops) Ciclos

Ahora que tenemos skills básicos en nuestro cinturón (Nota del autor: acaso los gatos usan cinturones?) podemos empezar a ser vagos. Como?! Sí, así es: Programar se trata de ser vago. Larry Wall, inventor del lenguaje de programación Perl, llamaba a la vagancia [**la mayor virtud**](http://c2.com/cgi/wiki?LazinessImpatienceHubris) de un buen programador. Si las computadoras no existiesen tendrías que hacer todo tipo de tareas tediosas a mano, pero si aprendes a programar puedes tirarte al sol todo el día mientras una computadora en algún lado corre los programas por vos. Es un estilo de vida glorioso lleno de relax!

Los ciclos son una de las formas más importantes de exprimir el poder de la computadora. Recuerdas Underscore.js de hace un rato? Asegúrate de haberlo cargado en la página (recuerda: siempre puedes usar la flechita para arriba un par de veces y apretar Enter para cargarlo de nuevo si lo necesitas) y trata de copiar y pegar esto en la consola:

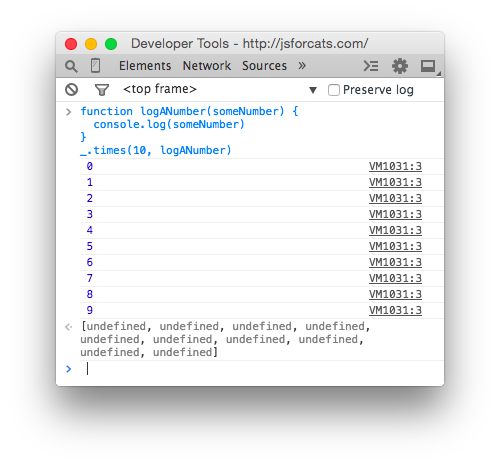
function imprimirNumero(unNumero) {

console.log(unNumero)

}

\_.times(10, imprimirNumero)

Este código usa el método **[times](http://underscorejs.org/" \l "times)** de Underscore que toma un número y una función, empieza en 0 y durante 10 pasos suma de a 1, llamando a la función con el número en cada paso.



Si tuvieramos que hacer manualmente lo que hace times se vería así:

imprimirNumero(0)

imprimirNumero(1)

imprimirNumero(2)

imprimirNumero(3)

imprimirNumero(4)

imprimirNumero(5)

imprimirNumero(6)

imprimirNumero(7)

imprimirNumero(8)

imprimirNumero(9)

Pero los gatos rechazan hacer trabajo manual innecesario como ese, entonces siempre debemos preguntarnos, "estamos haciendo esto del modo más vago posible?"

Entonces por qué se llama esto ciclado? Pensalo de esta manera: Si tuvieramos que escribir una lista de 10 números (de 0 a 9) usando un Array de JavaScript se vería como esto:

var ceroADiez = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Lo que times realmente hace es visitar cada número y repetir la tarea: en el ejemplo de arriba la tarea era llamar a la función imprimirNumero con el número actual. Repetir tareas de ese modo se llama ciclar sobre el Array.

### [#](https://jsparagatos.com/" \l "arrays) Arrays

Los mencioné unas cuantas veces pero pasemos un minuto aprendiendo sobre ellos. Imagina que necesitas hacer un seguimiento de tus amigos. Bueno, un Array lo va a hacer bien. Piensa en un Array como una lista ordenada donde puedes mantener una gran cantidad de cosas adentro.

Así es como haces uno:

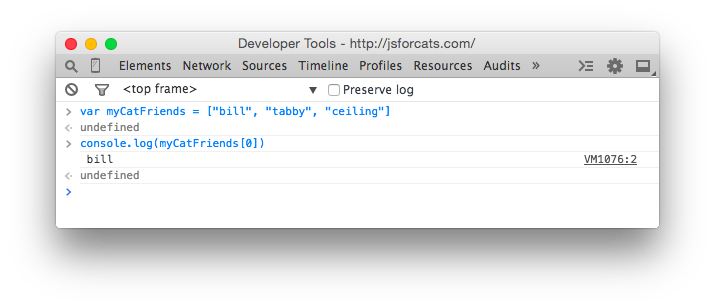
var misAmigosGatos = ["felix", "sessa", "garfield"]

Genial! Ahora tienes una lista de tus amigos gatos.

Los elementos (así es como llamas a un ítem de un Array) son guardados en arrays empezando por el 0 y contando para arriba. Entonces misAmigosGatos[0] devuelve felix y misAmigosGatos[1] devuelve sessa... etc etc.

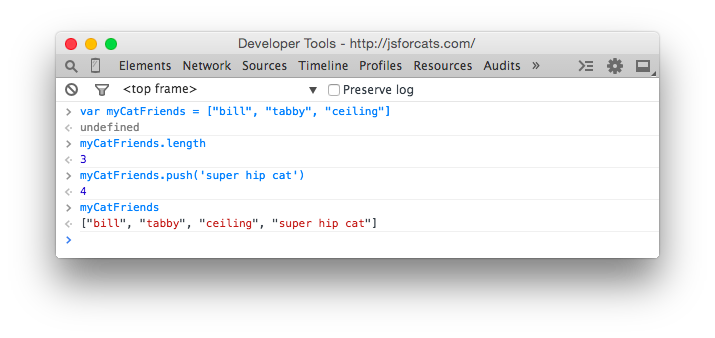
Para obtener un amigo de tu nuevo array puedes simplemente accederlo directamente así:

console.log(misAmigosGatos[0])



Si hiciste un nuevo amigo en el club gatuno de moda la otra noche y quieres agregarlo a tu lista es súper simple: misAmigosGatos.push("gato de moda").

Para verificar que el nuevo gato está adentro de tu array puedes usar .length:



Notaste que push devolvió el tamaño? Buenísimo! También toma nota que los arrays siempre preservan el orden lo que significa que van a recordar el orden en el que agregaste o definiste las cosas. No todo en JavaScript preserva el orden así que recuerda esta propiedad especial de los Arrays!

### [#](https://jsparagatos.com/" \l "objects) Objetos

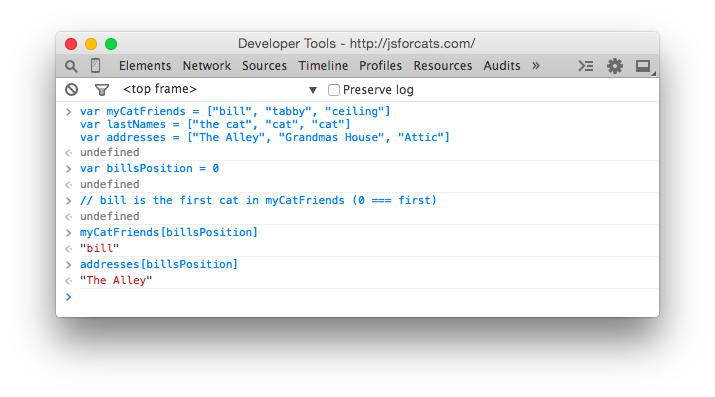
Los Arrays son buenos para listar, pero para otras tareas puede ser duro trabajar con ellos. Considera nuestro array de amigos felinos. Qué pasa si quieres guardar más que nombres?

var misAmigosGatos = ["felix", "sessa", "garfield"]

var apellidos = ["el gato", "gato", "gato"]

var direcciones = ["La calle", "Liniers", "La casa de John Bonachon"]

A veces es bueno tener todas las direcciones o nombres en una variable. Pero a veces tienes un gato en mente, digamos Félix, y querés buscar su dirección. Con arrays toma todo un trabajo porque no puedes decirle 'che array, dame la dirección de Félix' porque 'Félix' está en un array y su dirección en otro totalmente diferente.



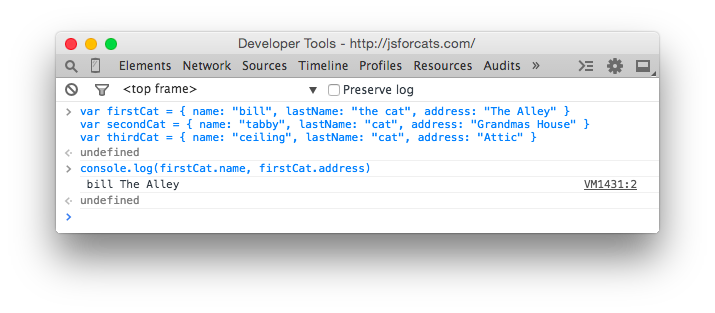
Esto puede ser frágil porque nuestro arrays cambian y agregamos un nuevo gato al comienzo podemos también actualizar la variable posicionDeFelix apuntando a la nueva ubicación de la información de Félix en los arreglos! Acá mostramos una forma más fácil de mantener guardada la información usando objetos:

var primerGato = { nombre: "felix", apellido: "el gato", dirección: "La calle" }

var segundoGato = { nombre: "sessa", lastName: "gato", direccion: "Liniers" }

var tercerGato = { nombre: "garfield", apellido: "gato", direccion: "Casa de John Bonachón" }

Por qué lo haríamos de esta manera? Porque ahora tenemos una variable para cada gato que podemos usar a la hora de obtener valores del gato de una manera más conveniente y legible.



Puedes pensar a los Objetos como llaves de un llavero. Cada una es para una puerta en particular y si tienes lindas etiquetas en tus llaves puedes abrir puertas muy rápidamente. De hecho, las cosas del lado izquierdo del : son llamadas claves (también se conocen como propiedades) y las cosas del lado derecho se llaman valores.

// un objeto con una sola clave llamada 'nombre' y un solo valor 'felix'

{ nombre: 'felix' }

Entonces por qué usar arrays si podemos poner toda nuestra información en objetos? Porque los objetos no recuerdan el orden de las claves que seteas. puedes ingresar un objeto de esta manera:

{ fecha: "20/10/2012", diario: "hoy dormí un poco", nombre: "Carlos" }

Pero la computadora puede devolverlo así:

{ diario: "hoy dormí un poco", nombre: "Carlos", fecha: "20/10/2012" }

O así!

{ nombre: "Carlos", diario: "hoy dormí un poco", fecha: "20/10/2012" }

Por lo tanto no puedes confiar en el orden de las claves de los objetos. Si querés hacer algo muy genial puedes hacer un array lleno de objetos o un objeto de arrays!

var archivoDeAnimo = [

{

fecha: "20/10/2012",

estado: "aburrido"

},

{

fecha: "21/10/2012",

estado: "dormido"

},

{

fecha: "22/10/2012",

estado: "divertido"

}

]

// ordenados de menos a más por favoritismo

var favoritos = {

golosinas: ["chocolate", "leche", "ratones"],

lugaresSiesta: ["sofa", "jardin", "pecho de humano"]

}

Cuando combinas diferentes cosas como esta estás haciendo estructuras de datos, como legos!

### [#](https://jsparagatos.com/" \l "callbacks) Callbacks

Los Callbacks no son realmente una característica de JavaScript como son los Objetos o los Arrays, sino una manera de usar funciones. Para entender por qué son útiles, primero tenemos que aprender sobre programación asíncrona (también llamada async). El código asíncrono es por definición escrito de una manera que no es síncrona. El código síncrono es fácil de entender y escribir. Este ejemplo lo ilustra:

var foto = descargar('http://foo-chan.com/images/sp.jpg')

subirTweetConFoto(foto, '@maxogden')

Este [**pseudo-código**](https://simple.wikipedia.org/wiki/Pseudocode) síncrono descarga una foto adorable de un gato, sube la foto a twitter y genera un tweet con la foto para @maxogden. Bastante simple!

(Nota del autor: Yo, @maxogden, acepto felizmente fotos de gatos por Twitter.)

Este código es síncrono porque para subir la foto al tweet, la descarga de la foto tiene que ser completada. Esto significa que en la linea 2 no puede correr hasta que la tarea de la linea 1 esté totalmente finalizada. Si fuéramos a implementar este pseudo-código queremos estar seguros de que descargar 'bloquea' la ejecución hasta que la descarga es finalizada, previniendo que cualquier otro JavaScript sea ejecutado hasta que termine, así cuando la descarga se completa, se desbloquea la ejecución de JavaScript y la linea 2 se ejecuta.

El código síncrono está bien para cosas que pasan rápido, pero es horrible para cosas que requieren guardar, cargar o descargar. Que pasa si el servidor del que estás bajando la foto es lento, o la conexión a internet que usas es lenta, o la computadora donde corres el código tiene muchas tabs con videos de gatos en YouTube que anda lento? Significa que potencialmente puede tardar minutos esperando hasta que la linea 2 se ejecute. Mientras tanto, como todo el JavaScript en la página está siendo bloqueado por la descarga, la página va a congelarse totalmente, quedando sin respuesta hasta que la descarga finalice.

La ejecución bloqueante debe ser evitada a todo costo, especialmente cuando hace que tu programa se congele o quede inusable. Asumamos que la foto de arriba toma 1 segundo para descargar. Para ilustrar cuanto es un segundo para una computadora moderna, acá hay un programa que testea cuantas tareas de JavaScript podemos procesar en un segundo.

function medirVelocidadCiclo() {

var contador = 0

function sumaUno() { contador = contador + 1 }

// Date.now() Retorna un número grande que representa la cantidad de

// milisegundos que pasaron desde el 1 de Enero de 1970

var ahora = Date.now()

// Cicla hasta que Date.now() es 1000 milisegundos (1 segundo) o más

// en el futuro respecto del comienzo del ciclado. En cada ciclo, llama a sumaUno

while (Date.now() - ahora < 1000) sumaUno()

// Finalmente pasaron >= 1000ms, entonces vamos a imprimir el total

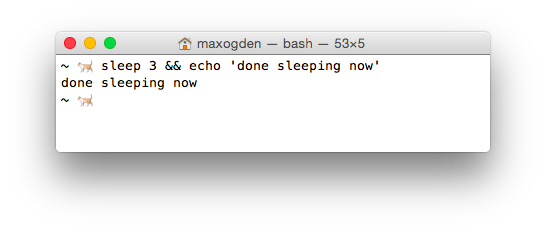
console.log(contador)

}

medirVelocidadCiclo()

Copia y pega el código de arriba en tu consola de JavaScript y luego de un segundo debería imprimir un número. En mi computadora veo 8527360, aproximadamente 8.5 millones. En un segundo JavaScript puede llamar sumaUno 8.5 millones de veces! Entonces si tienes código síncrono descargando una foto, y la foto tarda un segundo, significa que potencialmente estás previniendo la ejecución de 8.5 millones de operaciones mientras JavaScript está bloqueado.

Algunos lenguajes tienen una función llamada sleep que bloquea la ejecución por un número de segundos. Por ejemplo acá hay un código de [**bash**](https://en.wikipedia.org/wiki/Bash_(Unix_shell)) corriendo Terminal.app en Mac OS que usa sleep. Cuando corres el comando sleep 3 && echo 'me desperté' bloquea por 3 segundos antes de imprimir me desperté.



JavaScript no tiene una función sleep. Como sos un gato, seguramente te estás preguntando, "Por qué estoy aprendiendo un lenguaje de programación que no contempla dormir?". Pero no te vayas. En vez de depender de sleep para esperar a que pasen cosas, el diseño de JavaScript alienta el uso de funciones en su lugar. Si tienes que esperar que finalice la tarea A antes de comenzar la tarea B, pones todo el código de la tarea B en una función y la llamas solo cuando A terminó.

Por ejemplo, este es código bloqueante:

a()

b()

Y este es no bloqueante:

a(b)

En la versión no bloqueante, b es un callback de a. En la versión bloqueante a y b son los 2 llamados/invocados (los dos tienen () que ejecuta las funciones inmediatamente). En la versión no bloqueante vas a notar que solo a es invocada y b es simplemente pasada en a como un argumento.

En la versión bloqueante no hay una relación explícita entre a y b. En la versión no bloqueante, el trabajo de a es hacer lo que necesite y luego llamar a b cuando termina. Usar funciones de este modo se llama callbacks porque tu función de callback, en este caso b, es llamada cuando a termina de hacer lo que necesita.

Acá hay un pseudo-código con la implementación de como se debería ver a:

function a(fin) {

descargar('https://pbs.twimg.com/media/B4DDWBrCEAA8u4O.jpg:large', function finDescarga(error, png) {

// manejá el error si hay alguno

if (error) console.log('ups!', error)

// llamá a fin cuando termines

fin()

})

}

Volvamos a nuestro ejemplo no bloqueante, a(b), donde llamamos a a y le pasamos b como primer argumento. En la definición de la función a de arriba, el argumento fin es nuestra función b que le pasamos. Este comportamiento es difícil de entender al principio. Cuando llamas una función, los argumentos que pasas no van a tener necesariamente los mismos nombres de variable que cuando están dentro de una función. En este caso lo que llamamos b es fi dentro de la función. Pero b y fin son solo nombres de variables que apuntan a la misma función subyacente. Por lo general las funciones callback son etiquetadas con nombres como fin o callback para dejar claro que esas funciones deberían ser llamadas cuando la función actual termina su trabajo.

Entonces, mientras a haga su trabajo y llame a b cuando termine, ambas a y b son llamadas en sus versiones bloqueantes y no bloqueantes. La diferencia es que en la versión no bloqueante no bloqueamos la ejecución de JavaScript. En general el estilo no bloqueante es donde escribes toda función para que retorne lo más rápido posible, sin bloquear en ningún momento.

Para ir un poco más profundo: Si a toma un segundo para completar, y usas la versión bloqueante, significa que puedes hacer una sola cosa. Si usas la versión no bloqueante (usando callbacks) puedes hacer literalmente millones de otras cosas en el mismo segundo, lo que significa que puedes terminar tu trabajo millones de veces más rápido y dormir el resto del día.

Recordarlo: Programar se trata de ser vago y deberías ser vos el que duerma, y no tu computadora.

Con suerte puedes ver ahora como los callbacks son solo funciones que llaman a otras funciones luego de una tarea asíncrona. Algunos ejemplos clásicos de tareas asíncronas son leer una foto, descargar una canción, subir una foto, hablarle a la base de datos, esperar a que un usuario apriete una tecla o haga click en un link, etc. Cualquier cosa toma tiempo. JavaScript es genial manejando tareas asíncronas como esas siempre y cuando te tomes el tiempo de aprender como usar callbacks y mantener tu JavaScript lejos de ser bloqueado.

## Fin!

Este es solo el comienzo de tu relación con JavaScript! No puedes aprender todo de una, pero puedes buscar que funciona para vos e intentar aprender todos los conceptos acá.

Te recomiendo que vuelvas mañana y lo hagas todo de nuevo desde el principio! Puede tomarte algunas pasadas hasta que entiendas todo (programar es difícil). Solo intenta evitar leer esta página en cuartos que contengan objetos brillantes... Pueden ser increíblemente distractivos.

Tienes algún otro tópico que quieres que sea cubierto? Abre un issue para ello [**en github**](https://github.com/impronunciable/javascript-for-cats).

[#](https://jsparagatos.com/" \l "recommended-reading) Lectura recomendada

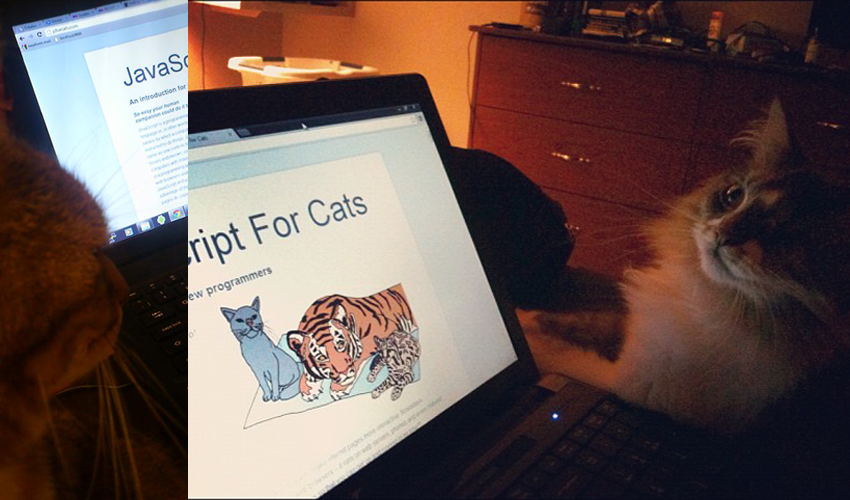
JavaScript para Gatos se saltea un montón de detalles que no son importantes para empezar (los gatos no son famosos por su atención), pero si sientes que necesitas aprender en profundidad mira esto:

Nota del traductor: La información de los siguientes links es en inglés. Lamentablemente escasea la bibliografía en español. Tomalo como una buena excusa para aprender un nuevo lenguaje (además de JavaScript)

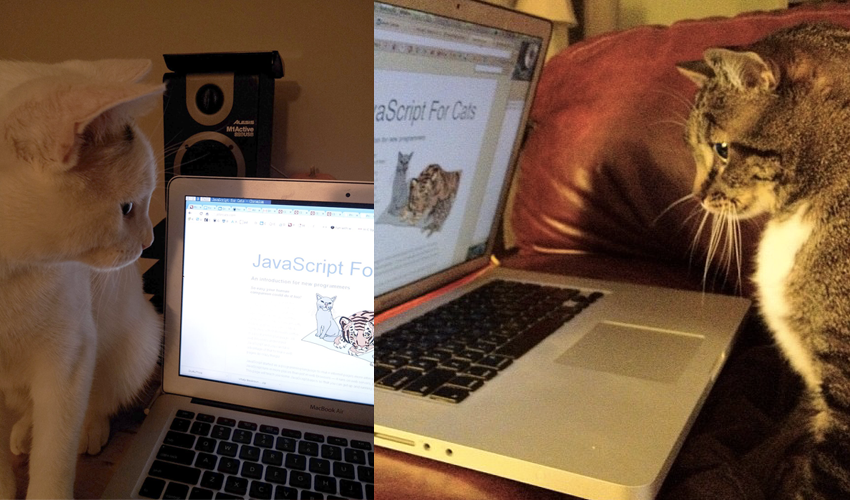
* [NodeSchool.io](http://nodeschool.io/) es un software educacional open source manejado por la comunidad que enseña diferentes áreas de web development en un formato interactivo y auto-guiado. Yo ayudé a hacer NodeSchool! lamentablemente este sitio tiene menos gatos.
* [Eloquent Javascript](https://eloquentjavascript.net/) es un libro gratuito que te enseña JavaScript! Es muy bueno! Especialmente el capítulo llamado [values, variables, and control flow](https://eloquentjavascript.net/chapter2.html)
* [La Guía de Mozilla de JavaScript](https://developer.mozilla.org/en-US/docs/JavaScript/Guide) también tiene un excelente capítulo llamado [values, variables and literals](https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Values,_variables,_and_literals)
* [La guía de estilo estándar` de JS](https://github.com/feross/standard) es un "zero configuration" linter para el estilo de JavaScript que yo uso.

### [#](https://jsparagatos.com/" \l "satisfied-customers) Clientes satisfechos











JSForCats.com es un trabajo en progreso con amor de [*@maxogden*](https://twitter.com/maxogden). Si quieres contribuir con este material y mejorarlo, el repo de GitHub está [*por acá*](https://github.com/maxogden/javascript-for-cats).



El traductor [**@impronunciable**](https://twitter.com/impronunciable) también es amigo de algunos gatos