Teoria Henry

Lección 1 - GIT

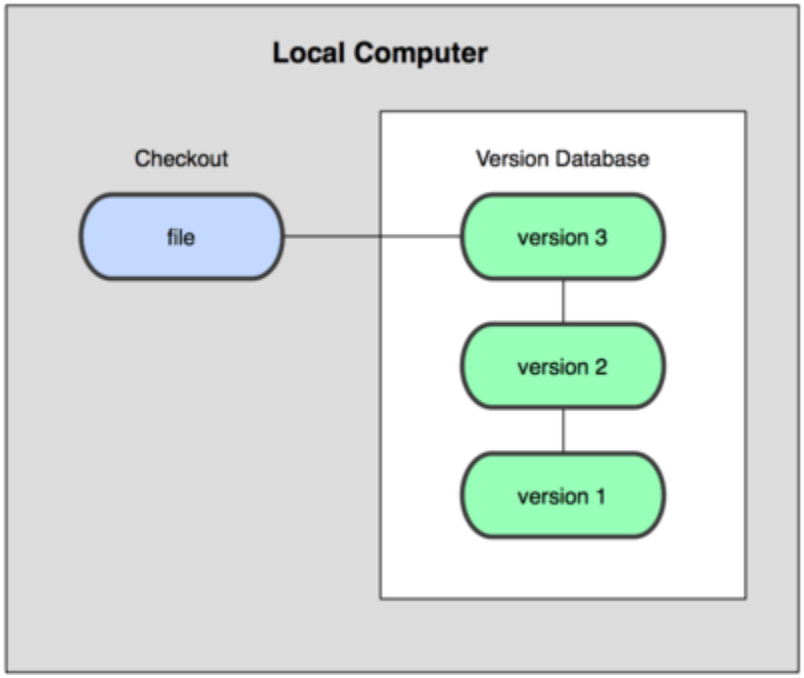
Version Control System

¿Qué es un control de versiones, y por qué debería importarte? Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Si eres diseñador gráfico o web, y quieres mantener cada versión de una imagen o diseño (algo que sin duda quieres), un sistema de control de versiones (Version Control System o VCS en inglés) es una elección muy sabia. Te permite revertir archivos a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Usar un VCS también significa generalmente que si rompes o pierdes archivos, puedes recuperarlos fácilmente.

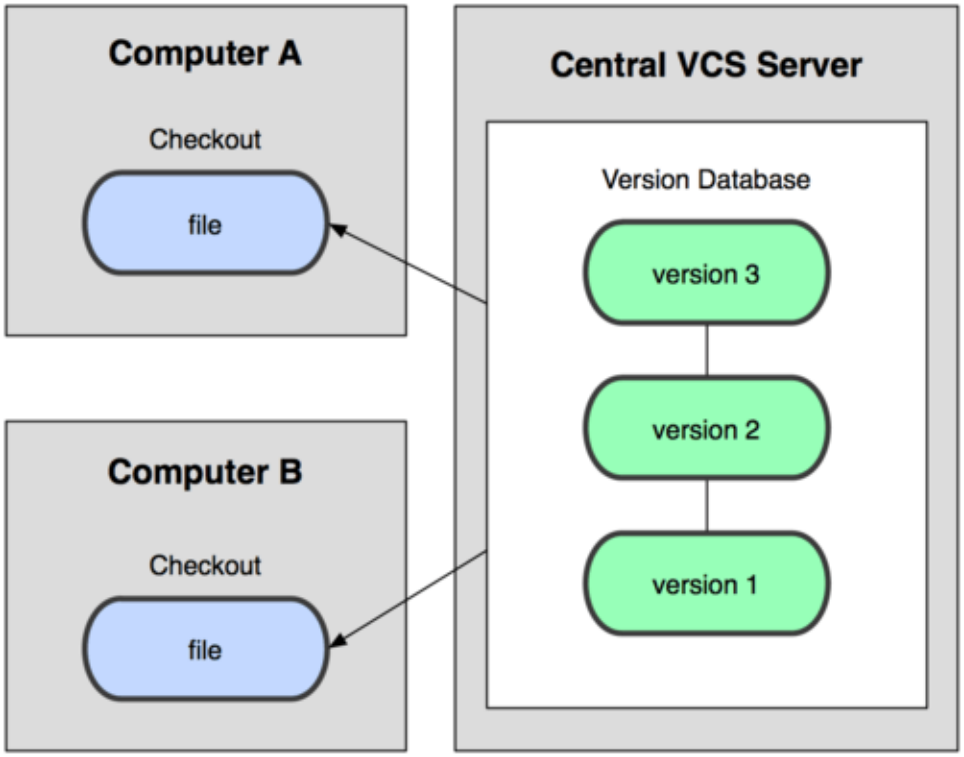
Hay varios tipos de sistemas de versionado, estos pueden ser:

Locales



Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías. Como se pueden imaginar, este sistema funciona *bien* para trabajar solos, pero si queremos incorporar otra gente al equipo van a empezar a surgir problemas.

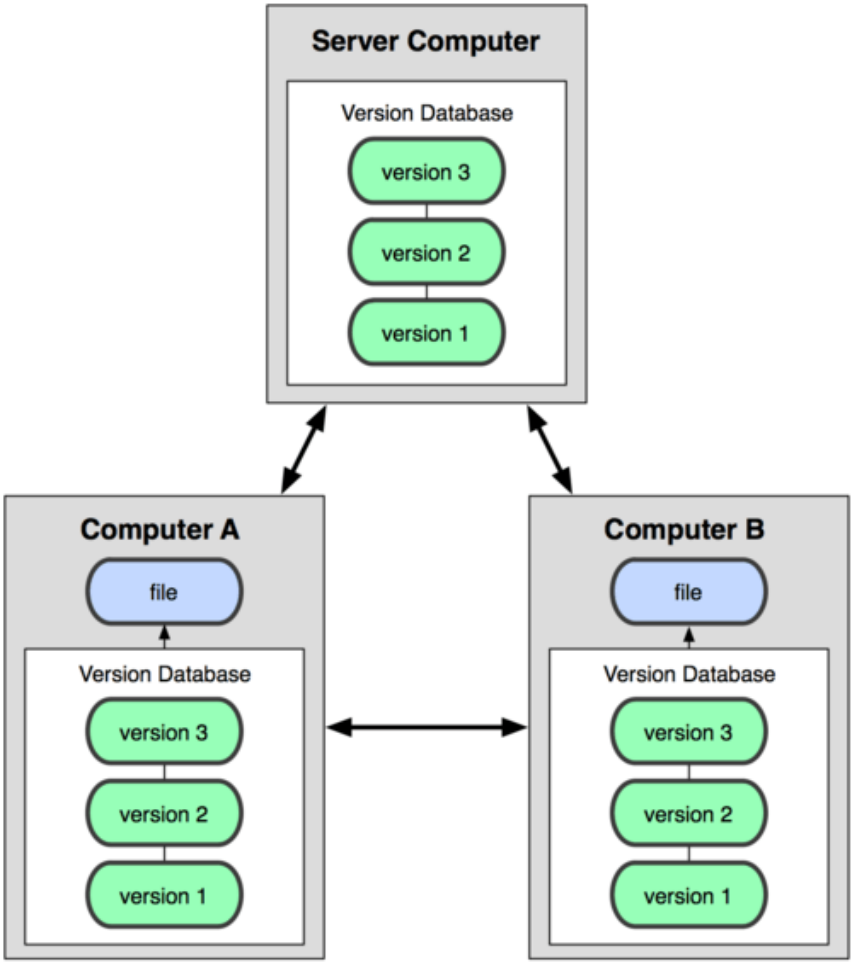
Centralizados



Para solventar este problema, se desarrollaron los sistemas de control de versiones centralizados (*Centralized Version Control Systems* o CVCSs en inglés). Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes descargan los archivos desde ese lugar central. Durante muchos años éste ha sido el estándar para el control de versiones.

Este sistema ofrece varias ventajas, como por ejemplo: Todo el mundo puede saber en qué están trabajando los demás colaboradores y los administradores tienen control sobre qué archivos pueden ver/modificar cada colaborador. Pero tambien presenta un *problema importante*: que hay un punto único de fallo. ¿Si éste server se cae? Nadie puede seguir trabajando ni trackeando sus cambios. ¿O si se rompe y no hay backups? Se pierde absolutamente *todo* el trabajo realizado.

Distribuido



Es aquí donde entran los sistemas de control de versiones distribuidos (*Distributed Version Control Systems* o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos: replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo.

Historia de Git

Como muchas de las grandes cosas en esta vida, Git comenzó con un poco de destrucción creativa y encendida polémica. El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de

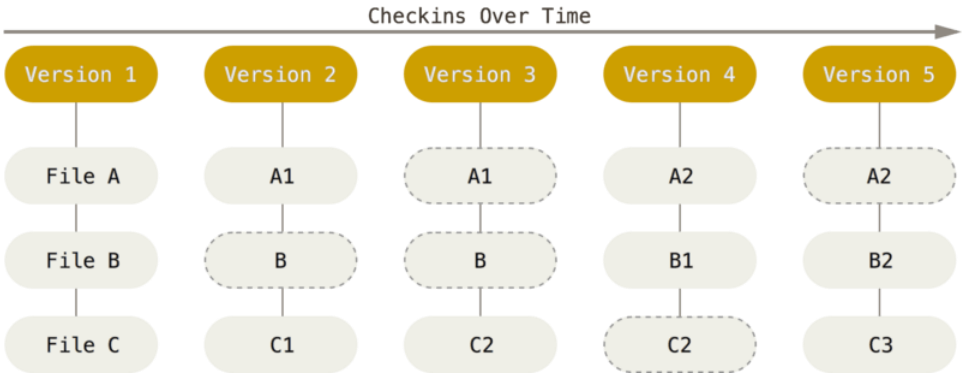
Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente a gran escala, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.

Conceptos de Git

Git modela sus datos como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.



La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas (con otros sistemas el proceso involucra llamados por red que generan retardos importantes).

Integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura de directorios. Un hash SHA-1 tiene esta pinta:

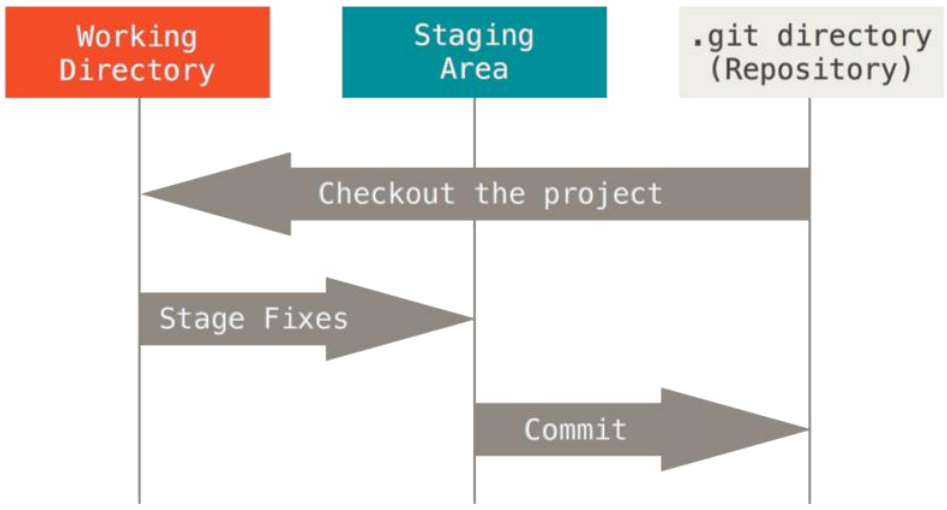
24b9da6552252987aa493b52f8696cd6d3b00373

Verás estos valores hash por todos lados en Git, ya que los usa con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Vamos a distinguir dos directorios, primero el *directorio de git*: que es donde almacena los metadatos y la base de datos de tu proyecto, y segundo el *directorio de trabajo* que es una copia de una versión del proyecto en particular. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar. Los archivos dentro del *directorio de trabajo* pueden estar en unos de los siguientes *estados*:

Estados
Git tiene tres estados principales en los que se pueden encontrar tus archivos:

- committed: significa que los datos están almacenados de manera segura en tu base de datos local.
- modified: significa que has modificado el archivo pero todavía no lo has commiteado a tu base de datos.
- staged: significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima commiteada.



Hay un archivo simple, generalmente contenido en tu directorio de Git, llamado que almacena información acerca de lo que va a ir en tu próxima confirmación, al contenido de este archivo. O al archivo mismo se lo conoce como staging area.

Sabiendo esto, el flujo de trabajo básico en Git sería algo así:

- Modificas una serie de archivos en tu *directorio de trabajo*.
- *Stageas* los archivos, añadiendolos a tu staging area o área de preparación.
- Commiteas o Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, y ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado (no se incluyó en el área de preparación), está modificada (modified).

Github.com

Github.com es una red para almacenar tus repositorios, esencialmente es un repositorio de repositorios. Es uno de los tantos disponibles en internet, y el más popular. Git != Github, aunque funcionen muy bien juntos. Github es un lugar donde puedes compartir tu código o encontrar otros proyectos. También actúa como portfolio para cualquier código en el que hayas trabajado. Si planeas ser un desarrollador deberías tener cuenta en Github. Usaremos Github extensivamente durante tu tiempo en Henry.

Lección 2: Introducción a Javascript

En esta lección cubriremos:

- Introducción a Javascript
- Variables
- Strings, Numbers y Booleanos
- Math
- Introducción a las Funciones
- Control de flujo y operadores de comparación
- Introducción a Node y NPM

<iframe src="https://player.vimeo.com/video/423852829" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Introducción a Javascript

JavaScript es un lenguaje de programación que fue creado originalmente para ser usado en el front-end de una pågina web. La idea original era poder dar dinamismo a las páginas webs, que en un principio eran estáticas. La introducción del "motor V8" de Google ha mejorado la velocidad y el funcionamiento de JS. Haciendo que JS (javascript) sea la lengua franca de la web, llegando inclusive al Back-End a través de NodeJs.

Vamos a aprender los conceptos más básicos de JS:

Variables

Una variable es una forma de almacenar el valor de algo para usar más tarde. (Una nota para aquellos con conocimientos previos de programación: Javascript es un lenguaje de tipado dinámico, una variable se puede configurar (y restablecer) a cualquier tipo, no necesitamos declarar su tipo al iniciar la variable).

Para crear una variable en JavaScript utilizamos la palabra clave `var`, seguida de un espacio y el nombre de la variable (con este nombre podremos hacer referencia a ella luego). Además de declarar una variable, podemos asignarle un valor usando el signo `=`.

Nota: Las palabras claves o keywords son palabras especiales que utiliza el lenguaje para indicar algo. No podremos usas las palabras claves del lenguaje cómo nombres de variables.

Existen tres formas de declarar una variable:

```
   var nombre = 'Juan'; // Vamos a usar principalemente esta forma    let apellido = 'Perez';    const comidafavorita = 'Pizza';
```

var

`var` es la forma declarar una variable en ES5 (ES5 es la versión de JS, hoy en día existe ES6 que es la nueva versión, pero que todavía no es la más usada). Esta es una *palabra clave* genérica para "variable".

Las dos formas siguentes, si bien son válidas, vamos a utilizarlas más adelante en la carrera, cuando tengamos más claros otros conceptos:

let

`let` es una nueva palabra clave de ES6, esto asignará una variable muy similar a `var`, pero con un comportamiento un poco diferente. Lo más notable es que difiere al crear un "nivel de *scope*" (hablaremos sobre esto más adelante).

const

`const` también es nuevo en ES6. Un `const` es una variable que no se podrá cambiar. Esto es la abreviatura de "constante".

console.log

Otro concepto del que hablaremos de inmediato es

```
console.log();
```

Este método muy simple nos permitirá imprimir en la consola todo lo que pongamos entre paréntesis.

Tipos de Datos

En ciencias de la computación, un tipo de dato informático o simplemente tipo, es un atributo de los datos que indica la clase de datos que se va a manejar. Esto incluye imponer restricciones en los datos, como qué valores pueden tomar y qué operaciones se pueden realizar.

Los tipos de datos aceptados varían de lenguaje en lenguaje.

Strings, Numbers, and Booleans

Estos son los tipos de datos más básicos en Javascript.

Strings

Las "strings" son bloques de texto, siempre se definirán entre comillas, ya sea simple o doble. Cualquier texto entre comillas es una cadena o string.

```
var nombrePerro = 'firulais';
```

Numbers

Los números son solo eso, números. Los números NO se envuelven en comillas. Pueden ser negativos también. Javascript tiene una limitación en el tamaño de un número (+/-9007199254740991), pero muy raramente aparecerá esa limitación en nuestro uso diario.

```
var positivo = 27;var negativo = -40;
```

Boolean

Los booleanos provienen de la lógica de Boole. Es un concepto que alimenta el código binario y el núcleo de las computadoras. Es posible que haya visto código binario en el pasado (0001 0110…), esto es lógica booleana. Esencialmente significa que tiene dos opciones, activar o desactivar, 0 o 1, verdadero o falso. En Javascript usamos booleanos para significar verdadero o falso. Esto puede parecer simple al principio, pero puede complicarse más adelante.

```
var meEncantaJavascript = true;
```

Los valores posibles de un dato booleando en JS son: `true` o `false`.

Operadores

Vamos a poder realizar operaciones en JavaScript a través de los `operadores`. Básicamente son símbolos que ya conocemos (`+`, `-`, `/`, `*`) que indican al intérprete de JavaScript las operaciones que debe realizar.

Por ejemplo: Para el intérprete al ver el signo `+`, sabe que tiene que ejecutar la función suma (que tiene internamente definida), y toma como parámetros los términos que estén a la izquierda y la derecha del operador.

```
var a = 2 + 3; // 5var b = 3 / 3; // 1
```

De hecho, esa forma de escribir tiene un nombre particular, se llama notación notación `infix` o `infija`, en ella se escribe el operador entre los operandos. Pero también existen otro tipos de notación como la `postfix` o `postfija` y la `prefix` o `prefija`. En estas última el operador va a la derecha de los operandos o a la izquierda respectivamente.

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

En fin, lo importante a tener en cuenta es que los operadores *son* funciones.

Precedencia de Operadores y Asociatividad

Esto parece aburrido, pero nos va a ayudar a saber cómo piensa el intérprete y bajo que reglas actua.

La *precedencia de operadores* es básicamente el orden en que se van a llamar las funciones de los operadores. Estás funciones son llamadas en *orden de precedencia* (las que tienen **mayor** precedencia se ejecutan primero). O sea que si tenemos más de un operador, el intérprete va a llamar al operador de mayor precendencia primero y después va a seguir con los demás.

La *Asociatividad de operadores* es el orden en el que se ejecutan los operadores cuando tienen la misma precedencia, es decir, de izquierda a derecha o de derecha a izquierda.

Podemos ver la documentación completa sobre Precedencia y Asociatividad de los operadores de JavaScript acá

Por ejemplo: `console.log( 3 + 4 * 5)` Para resolver esa expresión y saber qué resultado nos va a mostrar el intérprete deberíamos conocer en qué orden ejecuta las operaciones. Al ver la tabla del link de arriba, vemos que la multiplicación tiene una precedencia de 14, y la suma de 13. Por lo tanto el intérprete primero va a ejecutar la multiplicación y luego la suma con el resultado de lo anterior -> `console.log( 3 + 20 )` -> `console.log(23)`.

Cuando invocamos una función en Javascript, los argumentos son evaluados primeros (se conoce como **non-lazy** evaluation), está definido en la especificación. No confundir el orden de ejecución con asociatividad y precedencia, ver esta pregunta de StackOverflow.

Ahora si tuvieramos la misma precedencia entraría en juego la asociatividad, veamos un ejemplo:

```
var a = 1, b = 2, c = 3;a = b = c;console.log(a, b, c);
```

Qué veriamos en el console.log? Para eso tenemos que revisar la tabla por la asociatividad del operador de asignación `=`. Este tiene una precedencia de 3 y una asociatividad de `right-to-left`, es decir que las operaciones se realizan primero de derecha a izquierda. En este caso, primero se realiza `b = c` y luego `a = b` (en realidad al resultado de `b = c`, que retorna el valor que se está asignando). Por lo tanto al final de todo, todas las variables van a tener el valor `3`. Si la asociatividad hubiese al revés, todos las variables tendrían el valor `1`.

Math

Los operadores matemáticos trabajan en JavaScript tal como lo harían en su calculadora.

+ - * / =

```
1 + 1 = 22 * 2 = 42 - 2 = 02 / 2 = 1
```

%

Algo que quizás no haya visto antes es el Módulo (`%`), este operador matemático dividirá los dos números y devolverá el resto.

```
21 % 5 = 1;21 % 6 = 3;21 % 7 = 0;
```

Objetos Globales y métodos

Javascript tiene una serie de objetos integrados para que los usemos. Ya hemos visto, y hemos estado usando, el objeto de consola y su método `log`. Otro de estos objetos es `Math`. `Math` tiene varios métodos, al igual que `console` tiene `log`. Para agregar a esto, algunos de nuestros tipos de datos también tienen métodos incorporados.

Math.pow

Podemos usar el método `pow` en `Math` para devolver un número elevado a un exponente. Tomará dos números.

```
Math.pow(2,2) = 4;Math.pow(3,2) = 9;Math.pow(3,3) = 27;
```

Math.round , Math.floor, Math.ceil

`Math` también tiene métodos que redondearán los números para nosotros. `.round` redondeará un número al número entero más cercano. `.floor` siempre redondeará un número al número entero más cercano hacia abajo. `.ceil` siempre se redondeará al número entero más cercano hacia arriba.

```
Math.round(6.5) = 7;Math.round(6.45) = 6;Math.floor(6.999) = 6;Math.ceil(6.0001) = 7;
```

.length

El tipo de datos "string" tiene un método incorporado llamado `.length`. Cualquier cadena que llamemos a esto devolverá la cantidad de caracteres en esa cadena.

```
var nombreGato = 'felix';console.log(nombreGato.length); // 5
```

Veremos muchos otros métodos integrados en otros tipos de datos a lo largo de este curso.

Introducción a las Funciones

Las funciones son una parte muy importante de todo lenguaje de programacion y sobre todo en JavaScript. Són tipos particulares de Objetos, llamados `callable objects` u objetos invocables, por lo que tienen las mismas propiedades que cualquier objeto.

Ahora que tenemos un conjunto de variables, necesitamos funciones para calcularlas, cambiarlas, hacer algo con ellas. Hay tres formas en que podemos construir una función.

```
   function miFuncion() {}    var otraFuncion = function () {};    var yOtra = () =>
{};
```

Usaremos la primera forma en esta lección y hablaremos sobre las otras formas en próximas lecciones.

Anatomía de una Función

```
function miFuncion() {}
```

Una función comenzará con la palabra clave `function`, esto le dice a lo que sea que esté ejecutando tu programa que lo que sigue es una función y que debe tratarse como tal. Después de eso viene el nombre de la función, nos gusta dar nombres de funciones que describan lo que hacen. Luego viene un paréntesis abierto y uno cercano. Y finalmente, abra y cierre los corchetes. Entre estos corchetes es donde irá todo nuestro código a ejecutar.

```
function logHola() {    console.log('hola!');}logHola();
```

En este ejemplo declaramos una función `logHola` y la configuramos en `console.log 'hello'`. Entonces podemos ver que para ejecutar esta función, necesitamos escribir el nombre y los paréntesis. Esta es la sintaxis para ejecutar una función. Una función siempre necesita paréntesis para ejecutarse.

Argumentos

Ahora que podemos ejecutar una función básica, vamos a comenzar a pasarle argumentos.

```
function logHola(nombre) {    console.log('Hola, ' + nombre);}logHola('Martin');
```

Si agregamos una variable a los paréntesis cuando declaramos la función, podemos usar esta variable dentro de nuestra función. Iniciamos el valor de esta variable pasándola a la función cuando la llamamos. Entonces en este caso `nombre = 'Martin'`. También podemos pasar otras variables a esto:

```
function logHola(nombre) {    console.log( `Hola, ${nombre}`);}var miNombre =
'Antonio';logHola(miNombre);
```

Podemos agregar múltiples argumentos colocando una coma entre ellos:

```
function sumarDosNumeros(a, b) {  var suma = a + b;  return suma;}sumarDosNumeros(1,
5); // 6
```

Declaración "return" y Scope

En el ejemplo anterior presentamos la declaración `return`. No vamos a usar `console.log` con todo lo que salga de una función. Lo más probable es que queramos devolver algo. En este caso es la suma de los dos números. Piense en la declaración de retorno ("return") como la única forma en que los datos escapan de una función. No se puede acceder a nada más que a lo que se devuelve fuera de la función. También tenga en cuenta que cuando una función golpea una declaración de retorno, la función detiene inmediatamente lo que está haciendo y "devuelve" lo especificado.

```
function dividirDosNumeros(a, b) {  var producto = a / b;  return
producto;}dividirDosNumeros(6, 3); // 2console.log(producto); // undefined
```

Si intentamos `console.log` algo que declaramos dentro de la función, devolverá `undefined` porque no tenemos acceso a él fuera de la función. Esto se llama Scope ("alcance"). La única forma de acceder a algo dentro de la función es devolverlo.

También podemos establecer variables para igualar lo que devuelve una función.

```
function restarDosNumeros(a, b) {  var diferencia = a - b;  return diferencia;}var
diferenciaDeResta = restarDosNumeros(10, 9);console.log(diferenciaDeResta); //
1console.log(diferencia); // undefined
```

Podemos ver que la diferencia se establece dentro de la función. La variable dentro de la función solo pertenece dentro de la función.

Control de flujo y operadores de comparación

En este ejemplo, vamos a utilizar operadores de control de flujo y comparación. El flujo de control ("control flow") es una forma de que nuestra función verifique si algo es `true`, y ya sea ejecutando el código suministrado si es así o avanzando si no lo es. Para esto usaremos la palabra clave `if`:

```
function puedeManejar(edad) {    if (edad > 18) {        return true;    }    return
false;}puedeManejar(22); // true
```

Aquí estamos tomando un número (`edad`) y verificando si la declaración es `true` (`22>18`), lo es, por lo que devolveremos `true`, y la función se detendrá. Si no es así, omitirá ese código y la función devolverá `false`.

El símbolo "mayor que" (`>`) que ve en el último ejemplo se llama *Operador de comparación*. Los operadores de comparación evalúan dos elementos y devuelven `verdadero` o `falso`. Estos operadores son: `<`, `<=`, `>`, `> =`, `===`, `! ==`. Aprenderemos más sobre estos operadores en la próxima lección.

Introducción a Node y NPM

Node.js es un entorno de tiempo de ejecución desarrollado originalmente para su uso en servidores/back-end. Tendremos que instalarlo en nuestras máquinas para completar los próximos ejercicios. Para instalar Node, haga clic aquí: Descargar e instalar Node.js. Node viene con "NPM" incluido. NPM es un administrador de paquetes ("package manager") para paquetes Javascript y lo usaremos a lo largo de nuestro aprendizaje en Henry. Una vez que hayas instalado Node.js, no necesitas hacer nada más para instalar NPM.

Lección 3: Javascript II (Flujos de control, operadores de comparación, bucles `for`)

En esta lección cubriremos:

- `Undefined` y `null`
- Operadores de comparación (continuación)
- Flujos de control (continuación)
- Operados lógicos
- Bucles `for`
- `arguments`

<iframe src="https://player.vimeo.com/video/424318886" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Undefined y null

Hay un par de objetos Javascript que realmente no encajan en ningún tipo de dato. Esos son los valores `undefined` y `null`. Obtendrás `undefined` cuando busques *algo* que no existe, como una variable que aún no tiene un valor. `undefined` simplemente significa que lo que estás pidiendo no existe.

```
console.log(variableInexistente); // undefined
```

`null` es un objeto que nosotros, los desarrolladores, establecemos cuando queremos decirles a otros desarrolladores que el elemento que están buscando existe, pero no hay ningún valor asociado con él. Mientras que `undefined` está configurado por Javascript, `null` está configurado

por un desarrollador. Si alguna vez recibes `null`, debes saber que otro desarrollador estableció ese valor en `null`

```
let numeroTelefono = '11-1234-5678';numeroTelefono = null;numeroTelefono; // null
```

Una última cosa a tener en cuenta, ni `undefined` ni `null` son cadenas, están escritas tal como están sin comillas, como un booleano.

Veracidad

En estas lecciones hemos hablado sobre los valores booleanos, `true` y `false`. Cuando se usa una declaración `if` u otra declaración que espera un valor booleano (como `!`, *NOT*), y la expresión dada no es un valor booleano, Javascript hará algo llamado "coerción de tipo" y transformará lo que sea que se le entregue a un valor booleano. Esto se conoce como "truthy" y "falsey". Cada tipo de datos tiene una veracidad. Acá hay unos ejemplos:

```
// Datos que son forzados a verdaderos/"true"true1' '[] // Un array, aprenderemos más sobre esto más adelante{} // Un objeto, aprenderemos más sobre esto más adelantefunction() {}// Datos que son forzados a falsos/"false"false0undefinednull''
// Una cadena vacía
```

Operadores de comparación (continuación)

En la última lección usamos operadores de comparación, ahora profundizaremos un poco más sobre cómo funcionan y luego presentaremos un pariente cercano de operadores de comparación, los "operadores lógicos".

En la última lección presentamos nuestros operadores de comparación, (`> >= < <= === !==`). Estos operadores funcionan como lo harían en una clase de matemáticas, mayor que, menor que, etc. Utilizamos estos operadores para evaluar dos expresiones. A medida que la computadora ejecuta el código, el operador devolverá un `verdadero` (si la declaración es verdadera) o un `falso`.

```
1 > 2;     // false2 < 3;     // true10 >= 10;  // true100 <= 1;  // false
```

El "triple igual" (`===`) no debe confundirse con un solo signo igual (que indica asignar un valor a una variable). El triple igual comparará todo sobre los dos elementos, incluido el tipo, y devolverá si son exactamente iguales o no:

(Algo a tener en cuenta: hay un "doble igual" (`==`) que comparará dos elementos, pero NO tendrá en cuenta sus tipos (`1 == '1' // verdadero`). Debido a esto , se considera una mala práctica usar el doble igual. Nos gustaría verte siempre usando el triple, y siempre nos verás usándolo.)

```
1 === 1;          // true1 === '1';        // false'perro' === 'perro';  // true'perro' === 'Perro';  // false
```

El último operador de comparación que nos gustaría presentarle tiene dos partes.

Primero es el "NOT" (`!`). Cuando veas esto significará que estamos preguntando lo contrario de la expresión (volveremos a visitar el operador NOT más adelante en esta lección).

Con eso en mente, podemos introducir el "no es igual" (`!==`). Esto devolverá verdadero si los artículos NO son iguales entre sí de alguna manera. Esto, como el triple igual, tiene en cuenta el tipo de dato.

```
1 !== 1;          // false1 !== '1';        // true'perro' !== 'perro';  // false'perro' !== 'Perro';  // true
```

Flujos de control (continuación)

En la última lección aprendimos sobre el operador `if`. Podemos usar `if` para verificar y ver si una expresión es `true`, si es así, ejecute algún código, o si no es así, que omita el código y siga ejecutando el programa.

```
if (1 + 1 === 2) {    console.log('La expresión es verdadera');}
```

Para complementar a `if`, también podemos usar las declaraciones `else if` y `else`. Estas declaraciones deben usarse con `if` y deben venir después de él. Estas declaraciones serán evaluadas si el inicial `if` devuelve `false`. Podemos pensar en el `else if` como otra declaración `if` que se ha encadenado (podemos tener tantas otras declaraciones `if` que queramos). Solo se ejecutará un bloque de código de instrucción `if` o `else if`. Si en algún momento una declaración devuelve `true`, ese código se ejecutará y el resto se omitirá:

```
if (false) {    console.log('Este código será omitido');} else if (true) {    console.log('Este código correrá');} else if (true) {    console.log('Este código NO correrá');}
```

La declaración `else` siempre aparecerá al final de una cadena `if-else` o `if`, y actuará de manera predeterminada. Si ninguna de las expresiones devuelve `true`, el bloque de código `else` se ejecutará sin importar qué. Si alguna de las expresiones anteriores `if` o `else if` son `true`, el bloque de código de instrucción `else` no se ejecutará.

```
if (false) {    console.log('Este código será omitido');} else if (false) {    console.log('Este código NO correrá');} else {    console.log('Este código correrá');}
```

Operadores lógicos

También podemos combinar dos expresiones de igualdad y preguntar si alguna de las dos es verdadera, si ambas son verdaderas o si ninguna de ellas es verdadera. Para hacer esto, utilizaremos operadores lógicos.

&&

El primer operador lógico que veremos es el operador "Y" ("AND"). Está escrito con dos símbolos (`&&`). Esto evaluará ambas expresiones y devolverá verdadero si AMBAS expresiones son `true`. Si uno (o ambos) de ellos es falso, este operador devolverá `false`:

```
if (100 > 10 && 10 === 10) {    console.log('Ambas declaraciones son ciertas, este código se ejecutará');}if (10 === 9 && 10 > 9) {    console.log('Una de las declaraciones es false, por lo que && devolverá false, y este código no se ejecutará');}
```

||

El siguiente es el operador "Ó" ("OR"). Está escrito con dos barras verticales (||). Determinará si una de las expresiones es `true`. Devolverá `true` si una (o ambas) de las expresiones es `true`. Devolverá `false` si AMBAS expresiones son `false`:

```
if (100 > 10 || 10 === 10) {    console.log('Ambas declaraciones son ciertas, este código se ejecutará');}if (10 === 9 || 10 > 9) {    console.log('Una de las declaraciones es true, por lo que || devolverá true y este código se ejecutará');}if (10 === 9 || 1 > 9) {    console.log('Ambas declaraciones son falsas, por lo que || devolverá false y este código no se ejecutará');}
```

!

El último operador lógico es el operador "NOT" ("NO"). Está escrito como un solo signo de exclamación (`!`). Vimos este operador antes al determinar la igualdad (`!==`). Como antes, el operador NOT devolverá el valor booleano opuesto de lo que se le pasa:

```
if (!false) {    console.log('El ! devolverá true, porque es lo contrario de false, así que ste código se ejecutará');}if (!(1 === 1)) {    console.log('1 es igual a 1, de modo que la expresión devuelve true. El operador ! devolverá lo contrario de eso, por lo que este código se ejecutará');}
```

Notas sobre operadores lógicos

Un par de cosas a tener en cuenta sobre los operadores lógicos.

- Las expresiones se evalúan en orden, y la computadora omitirá cualquier expresión redundante. En una declaración `&&`, si la primera expresión es `false`, la segunda expresión no se evaluará porque AMBAS expresiones deben ser `true`. Lo mismo para la declaración `||`. Si la primera expresión es `verdadero`, la segunda no se evaluará porque solo debe haber una declaración `verdadero` para cumplir con los requisitos del operador.
- Usá paréntesis. Como vimos en el segundo ejemplo de operador `!`, usamos paréntesis para evaluar PRIMERO lo que estaba dentro de los paréntesis, luego aplicamos el operador `!`. Podemos ajustar cualquier expresión entre paréntesis y se evaluará antes de evaluar la expresión como un todo.

Bucles `for`

La mayoría del software se ejecuta en bucles, evaluando expresiones una y otra vez hasta que devuelve lo que estamos buscando o se detiene después de cierto tiempo. Javascript tiene dos expresiones de bucle incorporadas y hoy veremos la primera, el bucle "for".

Los bucles `for` tienen una sintaxis única, similar a la instrucción `if`, pero un poco más compleja. Primero tenemos la palabra clave `for`, seguida de paréntesis y luego abrir y cerrar llaves. Dentro

de los paréntesis necesitaremos tres cosas. Primero, debemos declarar una variable, esto es sobre lo que se repetirá el bucle. Entonces tendremos una expresión condicional, el ciclo continuará sucediendo hasta que esta declaración sea `false`. Tercero, incrementaremos nuestra variable. Las tres declaraciones están separadas por un punto y coma.

```
for (let i = 0                 ; i < 10                ; i++          ) {// |
Declaramos una variable | Expresión condcicional | Incrementamos la variable |
console.log(i);}
```

En este ejemplo, vemos que inicialmente establecemos nuestra variable `i` en 0, el ciclo se ejecutará y cada vez que llegue al final, aumentará el contador en uno. El bucle `for` evaluará la expresión condicional. Si es `true`, se ejecutará nuevamente, si es `false` dejará de funcionar.

El operador ++

Vimos en el último ejemplo el operador `++`. Esta es la abreviatura de Javascript para "Establecer el valor de la variable a su valor actual más uno". Hay algunas más de estas expresiones abreviadas de matemática / asignación variable, las visitaremos en las próximas lecciones.

Bucles infinitos

Es posible que un bucle se atasque en lo que llamamos un "bucle infinito". Debes asegurarte de que haya una forma de finalizar el bucle. Ejemplo de un bucle infinito:

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

Debido a que nuestra expresión condicional SIEMPRE será `true` (`i` nunca será menor que 0), este ciclo se ejecutará esencialmente para siempre. Esto interrumpirá su programa y puede bloquear su navegador web o computadora.

Arguments

Como vimos anteriormente, las funciones son objetos invocables, y podemos hacerlo pasándoles argumentos que varíen el comportamiento de estas.

```
> function log(str) {    console.log(str)  }> log('hola!')< 'hola!'
```

Si sabemos las variables a tomar, como en el ejemplo `str`, podemos darle nombre a este parámetro. Sino hay una propiedad ***arguments*** , propia de todas las funciones, que contiene los parámetros pasados como argumento.

```
> function args() {    console.log(arguments)  }> args('hola!', 'otro parametro', 3)<
["hola!", "otro parametro", 3, callee: 'function', Symbol(Symbol.iterator):
'function']
```

arguments nos da acceso a la **n** cantidad como parámetros, pero tengamos en cuenta que **no es un Arreglo**.

```
> function args() {    return Array.isArray(arguments)  }> args(1,2,3)< false
```

Si queremos saber cuantos parámetros puede recibir una función podemos usar la propiedad `length`.

```
> args.length< 0 // porque en la función `args` definimos 0 parámetros
```

Lección 4: Javascript III (continuación de bucles `for` y Arrays)

En esta lección cubriremos:

- Introducción a los arrays
- Bucles `for` con arrays

<iframe src="https://player.vimeo.com/video/424916422" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Introducción a los arrays (matrices/arreglos)
En la lección anterior discutimos los 3 tipos de datos básicos (cadenas/strings, números y booleanos) y cómo asignar esos tipos de datos a las variables. Discutimos cómo una variable solo puede apuntar a una sola cadena, número o booleano. Sin embargo, en muchos casos queremos poder apuntar a una colección de tipos de datos. Por ejemplo, ¿qué pasaría si quisiéramos hacer un seguimiento del nombre de cada estudiante en esta clase usando una sola variable, `nombresEstudiantes`. Podemos hacer eso usando Arrays. Podemos pensar en las matrices como contenedores de almacenamiento para colecciones de datos. Construir una matriz es simple, declarar una variable y establecerla en []. Luego podemos agregar al contenedor (separadas por coma) tantas cadenas, números o booleanos como queramos y acceder a esos elementos cuando lo deseemos.

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara', 'Samuel'];
```

.length
Al igual que el tipo de dato *String* tiene un método incorporado `.length`, también lo hace la matriz. De hecho, la matriz tiene muchos métodos incorporados útiles (los discutiremos en lecciones posteriores). Al igual que la cadena `.length` cuenta los caracteres, la matriz `.length` devolverá el número de elementos en una matriz:

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara',
'Samuel'];console.log(nombresEstudiantes.length);  // 4
```

Acceso a elementos en una matriz
Podemos acceder a un elemento de una matriza en cualquier momento, solo necesitamos llamar al elemento por su posición en la matriz. Los elementos reciben una posición numérica (índice) de acuerdo con su ubicación en la matriz, en orden. El orden numérico de una matriz SIEMPRE comienza en 0, por lo que el primer elemento está en el índice 0, el segundo en el índice 1, el tercero en el 2, y así sucesivamente (esto puede ser complicado al principio, pero solo recuerda que las matrices siempre comienzan en 0).

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara', 'Samuel'];
0         1         2         3
```

Para acceder al elemento, escribiremos el nombre o la variable de matriz, seguidos de corchetes que contienen la asignación numérica.

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara',
'Samuel'];console.log(nombresEstudiantes[1]);  // 'Antonio'
```

Para acceder dinámicamente al último elemento de la matriz, utilizaremos el método `.length`. En nuestra matriz `nombresEstudiantes`, la longitud es 4. Sabemos que el primer elemento siempre será 0, y cada elemento posterior se desplaza sobre un número. Entonces, en nuestro ejemplo, el último elemento tiene un índice de 3. Usando nuestra propiedad de longitud mostraremos cómo se hace cuando no sabemos el número de elementos en una matriz:

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara', ...
,'Samuel'];console.log(nombresEstudiantes[nombresEstudiantes.length - 1]);  //
'Samuel'
```

Asignación

Podemos asignar y reasignar cualquier índice en la matriz usando el paréntesis/índice y un "=".

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara',
'Samuel'];nombresEstudiantes[0] = 'Jorge';console.log(nombresEstudiantes);  //
['Jorge', 'Antonio', 'Sara', 'Samuel']
```

`.push` y `.pop`

Otros dos métodos de matriz incorporados muy útiles son `.push` y `.pop`. Estos métodos se refieren a la adición y eliminación de elementos de la matriz después de su declaración inicial.

`.push` agrega un elemento al final de la matriz, incrementando su longitud en 1. `.push` devuelve la nueva longitud.

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara',
'Samuel'];nombresEstudiantes.push('Patricia');console.log(nombresEstudiantes);  //
['Martin', 'Antonio', 'Sara', 'Samuel', 'Patricia']
```

`.pop` elimina el último elemento de la matriz, disminuyendo la longitud en 1. `.pop` devuelve el elemento "reventado" (*popped*).

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara',
'Samuel'];nombresEstudiantes.pop();console.log(nombresEstudiantes);  // ['Martin',
'Antonio', 'Sara']
```

`.unshift` y `.shift`

`.unshift` y `.shift` son exactamente como `.push` y `.pop`, excepto que operan en el primer elemento de la matriz. `.unshift(item)` colocará un nuevo elemento en la primera posición de la matriz, y `.shift()` eliminará el primer elemento de la matriz.

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara',
'Samuel'];nombresEstudiantes.unshift('Leo');console.log(nombresEstudiantes);  //
['Leo', 'Martin', 'Antonio', 'Sara',
'Samuel']nombresEstudiantes.shift();console.log(nombresEstudiantes);  // ['Martin',
'Antonio', 'Sara', 'Samuel']
```

Notas sobre las matrices

Debido a que Javascript no es un lenguaje fuertemente tipado, las matrices tampoco necesitan ser tipadas. Las matrices en Javascript pueden contener múltiples tipos de datos diferentes en la misma matriz

Utilizando bucles `for` en arrays

La mayoría de las veces, los bucles for se utilizan para iterar sobre todos los elementos de una matriz. Usando la técnica de acceso al índice ("index access technique") podemos acceder a cada elemento de la matriz. Para hacer esto, usamos el método `.length` como punto de parada para el ciclo.

```
const nombresEstudiantes = ['Martin', 'Antonio', 'Sara', 'Samuel'];for (let i = 0; i < nombresEstudiantes.length; i++) {    console.log(nombresEstudiantes[i]);}// 'Martin'// 'Antonio'// 'Sara'// 'Samuel'
```

Lección 5: Javascript IV (Objetos)

En esta lección cubriremos:

- Introducción a los Objetos
- Métodos
- Bucles `for…in`
- Palabra clave `this`
- Objetos en Javascript

<iframe src="https://player.vimeo.com/video/424936732" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Introducción a los Objetos

En la anterior lección aprendimos sobre *arrays* o matrices. Las matrices son contenedores que sostienen colecciones de datos. En esta lección, introduciremos otro contenedor de datos, el *Objeto*. Los objetos y las matrices son similares en ciertas cosas, y muy diferentes en otras. Mientras que los array pueden contener múltiples elementos relacionados unos con otros, los objetos contienen mucha información sobre una sola cosa. Los objetos se instancian usando llaves (`{}`).

```
const nuevoObjeto = {};
```

Pares Clave:Valor (`Key:Value`)

A diferencia de las matrices que tienen elementos valorados en índices, los objetos usan un concepto llamado pares de clave:valor. La clave (*key*) es el identificador y el valor (*value*) es el valor que queremos guardar en esa clave. La sintaxis es "clave: valor". Los objetos pueden contener muchos pares de clave-valor, deben estar separados por una coma (importante: sin punto y coma dentro de un objeto). Las claves son únicas en un objeto, solo puede haber una clave de ese nombre, aunque, varias claves pueden tener el mismo valor. Los valores pueden

ser cualquier tipo de dato de Javascript; cadena, número, booleano, matriz, función o incluso otro objeto. En esta demostración crearemos un objeto `usuario`.

```
const usuario = {    username: 'juan.perez',    password: 'loremipsumpwd123',
lovesJavascript: true,    favoriteNumber: 42};
```

Acceder a los valores

Una vez que tengamos los pares clave-valor, podemos acceder a esos valores llamando al nombre del objeto y la clave. Hay dos formas diferentes de hacer esto, usando puntos o usando corchetes.

Con la notación de puntos podemos llamar al nombre del objeto, un punto y el nombre de la clave. Así como llamamos a la propiedad `.length` en una matriz. La propiedad de longitud es un par de clave-valor.

```
user.lovesJavascript; // trueuser.username;        // juan.perez
```

La notación de corchetes es como llamar a un elemento en una matriz, aunque con corchetes debemos usar una cadena o número, o una variable que apunte a una cadena o número. Se puede llamar a cada clave envolviéndola con comillas:

```
const passString = 'password';user['lovesJavascript']; // trueuser['username'];
// juan.perezuser[passString];        // loremipsumpwd123
```

Generalmente, verás que los corchetes casi siempre se usan con variables.

Asignación de valores

Asignar valores funciona igual que acceder a ellos. Podemos asignarlos, cuando creamos el objeto, con notación de puntos o con notación de corchetes:

```
const nuevoUsuario = {    esNuevo: true}const loveJSString =
'lovesJavascript';nuevoUsuario.username =
'otro.nombre.de.usuario';nuevoUsuario['password'] =
'12345';nuevoUsuario[loveJSString] = true;
```

Eliminando propiedades

Si queremos eliminar una propiedad, podemos hacerlo usando la palabra clave `delete`:

```
const nuevoObjeto = {    eliminarEstaPropiedad: true};delete
nuevoObjeto.eliminarEstaPropiedad;
```

Es raro que veamos el uso de la palabra clave `delete`, muchos consideran que la mejor práctica es establecer el valor de una palabra clave en `undefined`. Dependerá de ti cuando llegue el momento.

Métodos

En los objetos, los valores se pueden establecer en funciones. Las funciones guardadas en un objeto se denominan métodos. Hemos utilizado muchos métodos hasta ahora a lo largo de este curso. `.length`, `.push`, `.pop`, etc., son todos métodos. Podemos establecer una clave para un nombre y el valor para una función. Al igual que otras veces que llamamos métodos,

llamaremos a este método usando notación de puntos y paréntesis finales (Nota: podemos llamar a un método con argumentos como lo haríamos con una función normal):

```
const nuevoObjeto = {    decirHola: function() {        console.log('Hola a todo el mundo!');    }}nuevoObjeto.decirHola(); //Hola a todo el mundo!
```

Bucles `for…in`

A veces queremos iterar sobre cada par clave-valor en nuestro objeto. Con las matrices, utilizamos un estándar para el bucle y una variable de número de índice. Los objetos no contienen índices numéricos, por lo que el bucle estándar no funcionará para los objetos. Javascript tiene un segundo tipo de bucle for integrado llamado "*for ... in loop*". Es una sintaxis ligeramente diferente, comienza igual pero entre paréntesis declararemos una variable, la palabra clave `in` y el nombre del objeto. Esto recorrerá cada clave del objeto y finalizará cuando se hayan iterado todas las claves. Podemos usar esta clave, y la notación de corchetes, en nuestro bucle for para acceder al valor asociado con esa clave.

```
const usuario = {    username: 'juan.perez',    password: 'loremipsumpwd123',    lovesJavascript: true,    favoriteNumber: 42};for (let clave in usuario){    console.log(clave);    console.log(usuario[clave]);}// username// 'juan.perez'// password// 'loremipsumpwd123'// lovesJavascript// true// favoriteNumber// 42
```

La palabra clave 'this'

Los objetos tienen una palabra clave autorreferencial que se puede aplicar en cada objeto llamado `this`. Cuando se llama dentro de un objeto, se refiere a ese mismo objeto. `this` puede usarse para acceder a otras claves en el mismo objeto, y es especialmente útil en métodos:

```
const usuario = {    username: 'juan.perez',    password: 'loremipsumpwd123',    lovesJavascript: true,    favoriteNumber: 42,    decirHola: function(){    console.log( this.username + ' manda saludos!');    }};usuario.decirHola(); // 'juan.perez manda saludos!'
```

Nota: la palabra clave `this` a veces puede ser uno de los temas más difíciles en Javascript. Lo estamos usando muy básicamente aquí, pero el tema se vuelve mucho más complejo muy pronto.

`this` y el Execution Context

Contexto global inicial

Este es el caso cuando ejecutamos código en el contexto global (afuera de cualquier función). En este caso `this` hace referencia al objeto `global`, en el caso del browser hace referencia a `window`.

```
// En el browser esto es verdad:> console.log(this === window);< true> this.a = 37;> console.log(window.a);< 37
```

En el contexto de una función

Cuando estamos dentro de una función, el valor de `this` va a depender de *cómo sea invocada la función*.

```
> function f1(){    return this;  }> f1() === window;< true> window.fi() === window;< true
```

En este ejemplo la función es invocada por el objeto global por lo tanto this hará referencia a `window`.

Si usamos el modo `strict` de Javascript, el ejemplo de arriba va a devolver `undefined`, ya que no le deja al interprete *asumir* que `this` es el objeto global.

Como método de un objeto

Cuando usamos el *keyword* `this` dentro de una función que es un método de un objeto, `this` toma hace referencia al objeto sobre el cual se llamó el método:

```
> var o = {    prop: 37,    f: function() {      return this.prop;    }  };> console.log(o.f());< 37// this hace referencia a `o`
```

En este caso, *no depende* donde hayamos definido la función, lo único que importa es que la función haya sido invocada como método de un objeto. Por ejemplo, si definimos la función afuera:

```
> var o = {prop: 37};// declaramos la función> function loguea() {    return this.prop;  }//agregamos la función como método del objeto `o`> o.f = loguea;> console.log(o.f());< 37// el resultado es le mismo!
```

De todos modos, hay que tener cuidado con el keyword `this`, ya que pueden aparecer casos donde es contra intuitivo (Como varias cosas de JavaScript). Veamos el siguiente ejemplo:

```
> var obj = {    nombre: 'Objeto',    log: function(){      this.nombre = 'Cambiado'; // this se refiere a este objeto, a `obj`      console.log(this)  // obj      var cambia = function( str ){        this.nombre = str;  // Uno esperaria que this sea `obj`      }      cambia('Hoola!!');      console.log(this);    }  }
```

Si ejecutamos el código de arriba, vamos a notar que después de ejecutar el código, la propiedad `nombre` de `obj` contiene el valor `Cambiado` y no `'Hoola!!'`. Esto se debe a que el keyword `this` dentro de la función cambia **NO hace referencia a `obj`**, si no que hace referencia al objeto global. No podemos considerar al `this` como obj porque la función no es método de este Objeto, fíjense que no podemos hacer obj.cambia. De hecho, si buscamos dentro del objeto global la variable `nombre`, vamos a encontrar con el valor `'Hoola!!'` que seteamos con la función `cambia`. Esto quiere decir que no importa en donde estuvo declarada la función, si no **cómo la invocamos**.

Prácticamente, no podemos saber a ciencia cierta que valor va a tomar el keyword hasta el momento de ejecución de una función. Porque depende fuertemente de cómo haya sido ejecutada.

Para resolver este tipo de problemas existe un patrón muy común, y se basa en guardar la referencia al objeto que está en `this` antes de entrar a una función donde no sé a ciencia cierta que valor puede tomar `this`:

```
var obj = {  nombre: 'Objeto',  log   : function(){    this.nombre = 'Cambiado'; //
this se refiere a este objeto, a `obj`    console.log(this); // obj    var that =
this; // Guardo la referencia a this    var cambia = function( str ){
that.nombre = str;  // Uso la referencia dentro de esta funcion    }
cambia('Hoola!!');    console.log(this);  }}
```

De esta forma, `that` (puede tener cualquier nombre) va a apuntar al objeto `obj` (`this` apuntaba a ese objeto cuando hicimos la asignación). Ahora si, podemos usar `that` en vez de `this` y estar seguros qué es lo que va a tener adentro.

Objetos en Javascript

En esta lección aprendimos qué son los Objetos y las muchas formas que existen para acceder a los valores, llamar a los métodos y asignar valores. Muchas de estas técnicas parecían muy familiares, como si las hubiéramos usado en prácticamente todos los aspectos de nuestros aprendizajes hasta ahora. Aquí hay un patrón, eso es porque TODO en Javascript es un Objeto. Las matrices son solo objetos con teclas numéricas, las cadenas son objetos bajo el capó con métodos incorporados, las funciones son en realidad objetos con sus propias propiedades especiales, todo el tiempo de ejecución de Javascript es un objeto (`window` en un navegador o `global` en el Node.js). Cuanto más trabajes con Javascript, más comenzará a tener sentido para ti. Solo recuerda, todo es un objeto.

Lección 6: Javascript V (Clases y `prototype`)

En esta lección cubriremos:

- Clases
- `prototype`

<iframe src="https://player.vimeo.com/video/425235994" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Clases

Muchas veces cuando creamos un objeto, estamos creando una plantilla. En lugar de copiar esa plantilla una y otra vez, Javascript nos da acceso a lo que llamamos un constructor o `class`. Las clases comparten gran parte de la misma funcionalidad que los objetos normales, pero también se expande mucho en esa funcionalidad. Las clases son útiles para crear muchos objetos que comparten algunas de las mismas propiedades y métodos (como los usuarios en un sitio web).

Class e instanciación pseudo-clásica

Si tienes experiencia en un lenguaje orientado a objetos (como Java o C#), probablemente estés familiarizado con el concepto de clases. Si bien Javascript no proporciona un "verdadero" sistema de clases, hay algo muy familiar. En aras de la discusión, llamaremos a nuestros objetos de clase 'clases'. Se instancia de manera pseudo clásica, usando la palabra clave `new`, y puede tomar argumentos.

En este ejemplo crearemos una clase `Gato`. La convención para las clases consiste en dar un nombre en mayúscula al nombre de todo lo que se pueda instanciar con la palabra clave `new`. Cuando usamos la palabra clave `new`, Javascript hace un gran trabajo detrás de escena para nosotros y crea y devuelve un objeto automáticamente.

```
function Gato(nombre) {    // El nuevo operador crea un objeto, "this"    this.nombre = nombre;    this.maullar = function() {        return 'Mi nombre es ' + this.nombre + ' ... Meow!';    }    // Devuelve el objeto "this"}const sam = new Gato('Sam');const kitty = new Gato('Kitty');console.log(sam.maullar()); // 'Mi nombre es Sam ... Meow!'console.log(kitty.maullar()); // 'Mi nombre es Kitty ... Meow!'
```

`this` en las clases

La palabra clave `this` puede comenzar a volverse muy confusa cuando comenzamos a usarla en clases. En el último ejemplo lo usamos en el método de los maullidos. Una buena regla general si no está seguro de a qué se refiere `this`, es observar dónde se llama el método y el objeto a la izquierda del 'punto'. Ese es el objeto al que se refiere `this`.

`prototype`

La creación de funciones es costosa (refiriéndonos a la capacidad de memoria de una computadora) y cada vez que creamos un nuevo objeto de clase con métodos, estamos recreando esos métodos en la memoria. Puede imaginar que si estamos creando miles de objetos de clase a partir de una clase con docenas de métodos, la memoria se acumulará rápidamente (20.000 - 40.000 métodos). Las clases tienen una forma única de establecer un método una vez y dar acceso a cada objeto de esa clase a esos métodos. Esto se llama el `prototype`. Cada clase tiene una propiedad *prototype*, que luego podemos establecer en métodos:

```
function Usuario(nombre, github) {    this.nombre = nombre;    this.github = github;}Usuario.prototype.introduccion = function(){    return 'Mi nombre es ' + this.nombre + ', mi usuario de Github es ' + this.github + '.';}let juan = new Usuario('Juan', 'juan.perez');let antonio = new Usuario('Antonio', 'atralice');console.log(juan.introduccion()); // Mi nombre es Juan, mi usuario de Github es juan.perez.console.log(riley.introduccion()); // Mi nombre es Antonio, mi usuario de Github es atralice.
```

Los métodos de `prototype` tienen acceso a la palabra clave `this` y, al igual que antes, siempre apuntará al objeto (a la izquierda del punto) que lo está llamando.

Hasta ahora siempre que teníamos que crear un objeto nuevo declarábamos un object literal, pero vamos a ver que hay otros métodos que nos da el prototype de Object para cumplir esa tarea

Object.create

El método `create` de los objetos nos permite crear un nuevo objeto a partir de un prototype especificado.

```
// creo un objecto con un objeto vacio como proto> var obj = Object.create({})> obj< Object {}// creo que un objeto a partir de un proto de Objeto> var obj =
```

```
Object.create(Object.prototype)// que es lo mismo que crear un objeto vacio literal> var obj = {}
```

Object.assign

El método `assign` de los objetos te permite agregar propiedades a un objeto pasado por parámetro

```
> var obj = {}// No hace falta guardar el resultado porque los objetos se pasan por `referencia`> Object.assign(obj, {nombre:'Emi', apellido:'Chequer'})> obj.nombre< 'Emi'
```

Herencia Clásica

En el paradigma de *Programación Orientada a Objetos* un tema muy importante es la *Herencia y Polimorfismo* y de las clases (los vamos a llamar constructores por ahora).

Cuando hacemos referencia a **Herencia** nos referimos a la capacidad de un constructor de *heredar* propiedades y métodos de otro constructor, así como un Gato es Mamífero antes que Gato, y hereda sus 'propiedades' (nace, se reproduce y muere).

Cuando hablamos de **Polimorfismo** nos referimos a la capacidad de que objetos distintos puedan responder a un llamado igual de acuerdo a su propia naturaleza.

Herencia en JavaScript

En JS a diferencia de la herencia clásica nos manejamos con prototipos, que van a tomar los métodos pasados por sus 'padres' mediante la `Prototype Chain`.

Cuando generamos un arreglo nuevo podemos acceder a métodos como `map` o `slice` gracias a que los heredamos del Objeto `Array` que esta vinculado en la propiedad `__proto__` y es el siguiente en el `Prototype Chain`.

Nosotros también podemos generar nuestros propios constructores que de los cuales heredar. Creemos un constructor de el cual pueda haber variantes.

```
> function Persona(nombre,apellido,ciudad) {    this.nombre = nombre; this.apellido = apellido;    this.ciudad = ciudad;  }> Persona.prototype.saludar = function() {    console.log('Soy '+this.nombre+' de '+this.ciudad);  }> var Emi = new Persona('Emi', 'Chequer', 'Buenos Aires');> Emi.saludar()< 'Soy Emi de Buenos Aires'
```

Ahora todo Alumno de Henry antes de Alumno es una Persona, asique podríamos decir que un Alumno hereda las propiedades de ser Persona.

```
> function Alumno(nombre,apellido,ciudad,curso) {    // podría copiar las mismas propiedades de Persona acá adentro    this.nombre = nombre;    this.apellido = apellido;    this.ciudad = ciudad;    this.curso = curso  }
```

Constructores Anidados

Pero en este caso estaríamos repitiendo código, y si en un futuro quisiera cambiar una propiedad tendría que hacerlo en ambos constructores. Descartemos esta opción.

```
// lo que nosotros queremos es poder reutilizar las propiedades de persona,> function Alumno(nombre, apellido, ciudad, curso) {    // usemos nuestro constructor Persona dentro del de Alumno    Persona.call(this, nombre, apellido, ciudad);    // vamos a necesitar el call porque queremos que las propiedades de persona, queden en bajo el objeto que va a devolver Alumno, y no uno nuevo del constructor Persona.    // luego le paso los valores que quiero que reciba el constructor de Alumno        // finalmente le agrego los puntos propios de Alumno    this.curso = curso;  this.empresa = 'Soy Henry';  }> var toni = new Alumno('Toni', 'Tralice', 'Tucuman', 'Web Full Stack')// Ahora si tenemos nuestra instancia creada a partir de ambos constructores> toni.curso< Web Full Stack> toni.apellido< Tralice> toni.saludar()< Uncaught TypeError: toni.saludar is not a 'function'// que paso?
```

Como podemos ver los métodos de *Personas* no fueron pasados a nuestros *Alumnos*. Veamos un poco el porqué.

El constructor del `__proto__` esta ligado a Alumno y luego al `Object Object` de JS. Pero el método `saludar` esta en el objeto `prototype` de Personas... , y esta perfecto, así es como debería funcionar, las instancias acceden al `__proto__` que fue vinculado por el constructor para ver que métodos tienen. Nuestro problema es que al llamar a Persona con `call` en vez de con el método `new` no se esta haciendo ese vinculo en el que `Persona.prototype` se mete en nuestro `Prototype Chain`, y entonces las instancias de Alumno no tienen acceso a los métodos de Persona

Vamos a solucionar ese problema agregando al prototipo los métodos de Persona, para esto vamos a usar el método `Object.create`.

```
// usamos `Object.create` porque este guardaba el argumento pasado como `__proto__` del objeto a retornar> Alumno.prototype = Object.create(Persona.prototype)// si recuerdan el objeto prototype siempre tenia una propiedad constructor que hacia referencia a la función en si, con la asignación que hicimos arriba lo pisamos, por lo que deberíamos volver a agregarlo.> Alumno.prototype.constructor = Alumno> var Franco = new Alumno('Franco','Etcheverri','Montevideo','Bootcamp')> Franco.saludar()< 'Soy Franco de Montevideo'
```

Lección 7: Javascript VI (Callbacks)

En esta lección cubriremos:

- Callbacks
- Más métodos de Arrays
- Introducción a la programación funcional

<iframe src="https://player.vimeo.com/video/425254623" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Callbacks

Un concepto muy importante en Javascript es la capacidad de pasar una función como argumento a otra función. Estas funciones se denominan `callbacks`. Estas funciones pueden llamarse en cualquier momento y pasar argumentos dentro de la función. Pronto

descubriremos por qué las devoluciones de llamada son tan importantes para Javascript. La convención es usar `cb` como argumento para la variable que se usará de callback.

```
function decirHolaAlUsuario(usuario) {    return 'Hola ' + usuario + '!';}function decirAdiosAlUsuario(usuario) {    return 'Adiós ' + usuario + '!';}function crearSaludo(usuario, cb) {    return cb(usuario);}crearSaludo('Dan', decirHolaAlUsuario); // 'Hello Dan!'crearSaludo('Dan', decirAdiosAlUsuario); // 'Goodbye Dan!'
```

Más métodos de Arrays

Ya conocemos y utilizamos métodos de matriz, `.push`, `.pop`, `.shift`, `.unshift` y `.length`. Pero hay muchos más métodos disponibles de forma nativa en un array. Los métodos de los que vamos a hablar aquí se denominan "métodos de orden superior", porque toman los callbacks como argumentos.

`.forEach`

`.forEach` es un bucle for integrado en cada array. `.forEach` toma un callback como su único argumento, e itera sobre cada elemento de la matriz y llama al callback en él. El callback puede tomar dos argumentos, el primero es el elemento en sí, el segundo es el índice del elemento (este argumento es opcional).

```
const autos = ['Ford', 'Chevrolet', 'Toyota', 'Tesla'];// Podemos escribir el callback en los paréntesis como una función anónimaautos.forEach(function(elemento, indice) {    console.log(elemento);});// O podemos crear una instancia de una función para usarla como callback.// Además, no necesitamos usar el argumento de índice, si no lo necesitas, no dudes en omitirlo.function mostrarNombres(elemento) {console.log(elemento);}// And call that function in the forEach parenthesesautos.forEach(mostrarNombres);
```

`.reduce`

`.reduce` ejecutará un bucle en nuestra matriz con la intención de reducir cada elemento en un elemento que se devuelve. Como es el primer argumento, acepta un callback que toma dos argumentos, primero un 'acumulador' (el resultado del método de reducción hasta ahora), y el segundo es el elemento en el que se encuentra actualmente. El callback debe contener siempre una declaración de devolución ("return"). `.reduce` también toma un segundo argumento opcional, que sería el acumulador de arranque ("starting accumulator"). Si no se suministra el acumulador de arranque, la reducción comenzará en el primer elemento de la matriz. `.reduce` siempre devolverá el acumulador cuando termine de recorrer los elementos.

```
const numeros = [ 1, 2, 3, 4, 5, 6, 7, 8, 9];const palabras = [ 'Hola,', 'mi', 'nombre', 'es', 'Martin'];// Podemos escribir la función anónima directamente en los paréntesis de .reduce// Si omitimos el elemento inicial, siempre comenzará en el primer elemento.const suma = numeros.reduce(function(acc, elemento){    return acc + elemento;});// Podemos escribir una función fuera de los parents de .reduce (para usar varias veces más tarde)function multiplicarDosNumeros(a, b) {    return a * b;}const productos = numeros.reduce(multiplicarDosNumeros);// .reduce funciona en cualquier tipo de datos.// En este ejemplo configuramos un acumulador de arranqueconst frases = palabras.reduce(function(acc, elemento) {    return acc + ' '
```

```
+ elemento;}, 'Frase completa:');console.log(suma); // 45console.log(productos); //
362880console.log(frases); // "Frase completa: Hola, mi nombre es Martin"
```

`.map`

`.map` se usa cuando queremos cambiar cada elemento de una matriz de la misma manera. `.map` toma una devolución de llamada como único argumento. Al igual que el método `.forEach`, el callback tiene el elemento y el índice de argumentos opcionales. A diferencia de `.reduce`, `.map` devolverá toda la matriz.

```
const numeros = [2, 3, 4, 5];function multiplicarPorTres(elemento) {    return
elemento * 3;}const doble = numeros.map(function(elemento) {    return elemento *
2;});const triple = numeros.map(multiplicarPorTres)console.log(doble); // [ 4, 6, 8,
10 ]console.log(triple); // [ 6, 9, 12, 15 ]
```

Lección 8: Fundamentos HTML/CSS

En esta clase veremos:

- Introducción a HTML.
- Elementos/tags HTML básicos.
- Introducción a CSS.
- Selectores CSS y el tag <style> .
- Estilos Básicos.
- Modelo de Caja.
- Hojas de estilos externas y el tag <link>.

Introducción HTML

HTML es el bloque básico con el que está construido internet. Todas las páginas web utilizan HTML. *No es un lenguaje de programación* propiamente dicho, sino, es un lenguaje de **Markup**: son lenguajes que incorporan al texto marcas o etiquetas que luego son interpretadas para darle información extra sobre la estructura del texto. En el caso de HTML, este será interpretado por los browsers, que tambien presentaran el código en forma gráfica.

HTML es la abreviatura de **Hyper Text Markup Language**:

- Hyper Text: "Hyper Texto" quiere decir texto con links
- Markup Language: Los "Lenguajes de Marcado" son lenguajes de programación basados en etiquetas que uno agrega a un texto para darle estructura e información adicional. A diferencia de los "Lenguajes de Scripting" que se usan para crear programas informáticos, los lenguajes de marcado son sólo reglas para ordenar un documento.

Elementos básicos HTML

HTML define una serie de elementos (o etiquetas, o tags) que serviran para delimitar texto. Cada tag está encerrado en `< >` y tiene un nombre. Los tags se abren y se cierran, los tags de cerrado incluyen con un "/" en el principio del tag que cierra. Por ejemplo:

```
<element>    ...  </element>
```

Algunos tags html, por su naturaleza, no necesitan tener nada *adentro*. Por lo tanto podemos abreviar su escritura y en vez de abrir y cerrar el tag, simplemente agregamos un "/" antes del bracket final.

```
<img src="http://imagen.com/img.jpg" />
```

Atributos

En su mayoría de los atributos de un elemento son pares *nombre-valor*, separados por un signo de igual «=» y escritos en la etiqueta de comienzo de un elemento, después del nombre del elemento. El valor puede estar rodeado por comillas dobles o simples. Los atributos de los tags nos sirven para cambiar su comportamiento o *configurarlos*.

Por ejemplo, el tag `<img>` sirve para mostrar una imagen. Este tag recibe el atributo `src` que indica la (URL)[https://es.wikipedia.org/wiki/Localizador_de_recursos_uniforme] de donde está la imagen que queremos mostrar.

```
<img src="http://imagen.com/img.jpg" />
```

<html>

El tag `<html>` va a contener a todos los demás tags dentro suyo. Este tag básicamente sirve para avisarle al browser que el contenido debe ser interpretado como `html`.

<head>

Este tag sirve para contener tags que contengan información sobre el documento, pero es información que no queremos que se renderee. Comunmente contiene el *titulo* de la página y *links* a recursos externos que pueda usar la página (javascript o css).

<title>

Es el título de la página, se mostrará en el tab del browser o en la parte superior (pero no en la página).

<body>

En este tag estára encerrado todo lo que querramos que se vea en la pantalla.

Entonces, hasta ahora, un documento HTML se ve así:

```
<html>    <head>      <title>Es el título de nuestra página</title>    </head>
<body>    </body>  </html>
```

Como ven, para mayor fácilidad en la lectura y la estructuración del documento, el documento HTML se escribe indentando (o usando sangría).

(Todos los tags que presentaremos más abajo van siempre adentro de un tag `<body>`)

<p>
Es el tag para los párrafos. Mostrará el texto contenido dentro en una nueva línea.

```
<p>Soy un párrafo</p>
```

****
El elemento span es un contenedor de texto genérico. No inserta una nueva linea, como lo hace el elemento p. Sirve básicamente para darle estilo al texto.

** **
El elemento div es un *contenedor* genérico. Es usado principalmente para dar estilo, imaginen que es una caja (cuyo tamaño y color podes modificar *a piacere*), y que dentro podés poner otras cajas iguales.

<a>
El tag a (del inglés *anchor*), nos permite crear **links** a otros documentos y páginas. Este tag recibe el *atributo* `href` que indica a dónde apunta el link.

```
<a href="http://www.soyhenry.com">Esto es un link!</a>
```

<h1> ... <h6>
Son los tags de encabezado o títulos, están pensandos del 1 al 6, para indicar la importancia del contenido y su jerarquía.

```
<h1>El título más importante!</h1> <h3>título medianamente importante.</h3> <h6>El título menos importante.</h6>
```

<img>
Este tag nos permite mostrar imágenes en la pantalla. Necesita el atributo `src` que indica la *URL* de donde sacar la imagen a mostrar.

```
<img src="http://www.soyhenry.com/imageurl.jpg" />
```

<ul>
Este tag representa una lista desordenada (del inglés "unordered list"). Este tag está diseñado para contener otros tags de tipo item. También existe el tag `<ol>` que viene de "ordered list".

<li>
Son los tags que contienen los items de la lista ('list item').

```
<ul> <li> <span>Elemento uno</span> </li> <li> <p>Podemos anidar cualquier tipo de tag adentro</p> </li> <li> <span>tercer elemento</span> </li> </ul>
```

CSS

Como vimos, HTML nos sirve para dar estructura al contenido. En las primeras épocas de internet las páginas eran así. De hecho, todavía esta online la primera página web. Como ven es bastante aburrida. Luego se introdujo el concepto de CSS (Cascading Style Sheets); una forma de poder agregar color y estilos en nuestras páginas!

Reglas CSS

Básicamente una regla CSS está compuesta por un atributo o propiedad y un valor. Seǵun el atributo que usemos y el valor que le pongamos a ese atributo vamos a obtener resultados visuales distintos en nuestro html.

Por ejemplo:

```
html {  color: red; /* "Color" es la propiedad y "red" el valor */  font-size: 12px;
/* "font-size" es la propiedad y "12px" el valor */}
```

En este ejemplo, vemos dos atributos: `color` y `font-size`, el primero permite modificar el color del texto, en este caso está seteado a `red`; el segundo indica el tamaño del texto, en este caso `12px`. Es importante notar que distintos atributos pueden recibir distintos valores, generalmente los que indican un color reciben un color (`red, blue`, etc...), los que son medidas reciben una medida (`12px, 15px`, etc..), y hay otras propiedades que reciben valor especificos, por ejemplo: la propiedad `border` (que dibuja un borde alrededor de un elemento) recibe tres valores: el color del borde, el ancho de la linea y el tipo de linea (punteada, continua, etc..).

Hay muchos atributos CSS disponibles, más de los que podemos recordar. Así que no se asusten, con el tiempo van a empezar a memorizar estas propiedades. Pueden ver una lista completa acá.

Formas de dar estilo

Antes de empezar a dar estilos, necesitamos una forma de decirle al browser qué vamos a darles reglas de estilo. Hay varias formas de lograr esto (más adelante veremos en detalle como funcionan cada una):

- usando el atributo style: esta es la forma primitiva más simple, básicamente le damos reglas a cada tag html.
- usando el tag `<style/>`: Se utiliza este tag en el `<head>` del documento HTML, con esto logramos agrupar todas las reglas que luego queremos que se apliquen a los elementos HTML.
- Usar el tag `<link/>`: Este método nos permite definir las reglas CSS en un documento separado e *importarlo* a nuestra página (la ventaja que tiene es que podemos importar el mismo CSS a varias páginas).

Atributo style

Todos los *tags* HTML pueden recibir el atributo `style`. Este atributo indica las reglas CSS (que veremos más abajo), que se aplicaran **sólo** al elemento que las tiene.

```
<h1 style="color:blue;">Esto es un título Azul</h1>
```

Pros:

- Fácil de escribir y leer.
- Cómo se aplican a un sólo elemento no hay forma de confundirse y que se aplique la regla a un elemento no deseado. Cons:
- La regla aplica a un sólo elemento, si quisieramos que varios elementos tengan la misma regla, deberíamos copypastear!

<style/>

El tag *style*, que se escribe en el del documento, nos permite escribir reglas que se aplicaran a uno o varios elementos html. Es importante notar que con esta forma, podremos darle estilo a muchos elementos de una sóla vez, pero sólo a elementos que estén en el mismo documento.

```
 <html>    <head>      <style>        /*<!-- acá van las reglas -->*/      </style>
</head>    <body>    </body>  </html>
```

Pros:

- Lugar central donde podemos escribir las reglas CSS del documento
- Podemos compartir reglas entre varios elementos iguales Cons:
- No podemos compartir las reglas con *otro* documento HTML.
- Hay que prestar atención a las reglas, y a qué elementos se aplican.

Con el tag `<link>` dentro del `<head>` del documento, vamos a poder *importar* un archivo css que contenga varias reglas CSS. Funciona similar al tag `<style/>` anterior. Pero ahora tenemos la ventaja que podemos *compartir* el mismo archivo css con varios documentos HTML.

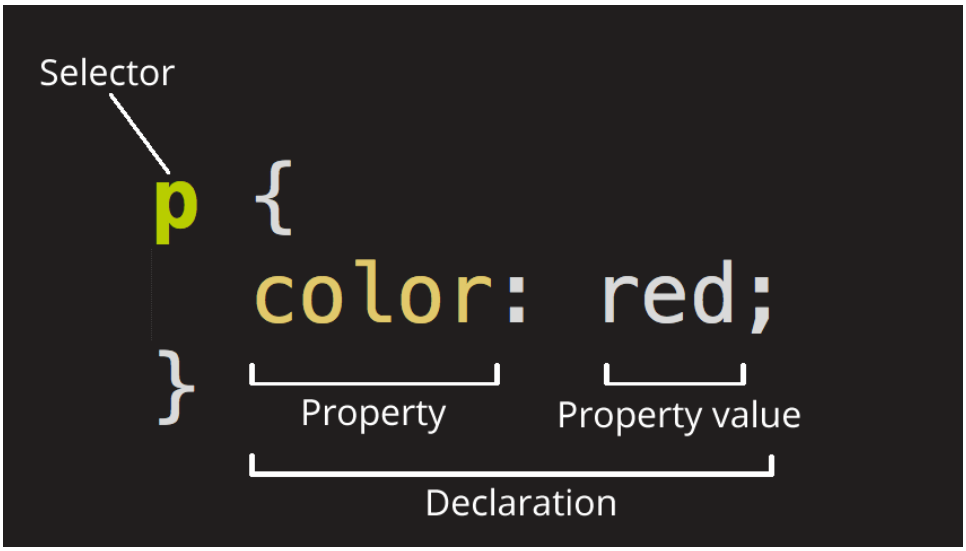
```
<!DOCTYPE html><html><head>  <link rel="stylesheet"
href="styles.css"></head><body></body></html>
```

Pros:

- Lugar central donde podemos escribir las reglas CSS del documento
- Podemos compartir reglas entre varios elementos iguales
- Podemos compartir reglas entre varios documentos HTML Cons:
- Hay que prestar atención a las reglas, y a qué elementos se aplican.

Selectores

Para poder aplicar reglas de estilo a los elementos html, necesitamos una forma de saber cómo seleccionar los elementos a los que deseamos aplicar las reglas, para esto sirven los *selectores CSS*.



Hay varios tipos de selectores, los más básicos son los de tipo, donde indicamos a qué clase de elemntos se van a aplicar las reglas, el ejemplo de arriba usa un selector de tipo. Está diciendo: *aplicarle a todos los elementos de tipo <p/> la regla de texto color rojo*.

El selector de tipo se puede usar con cualquier tipo de tag: p, div, body, etc. Otra forma de usar selectors poniendole un *nombre* o *identificador* a cada elemento HTML. Para esto existe un `atributo` que pueden recibir todos los tags llamados: `id` y `class`.

```
<div id="divId"></div>  <div class="divClass"></div>
```

Ids: son nombre que sólo pueden aparecer una sólo vez en el documento. Es super especifico y sirve para seleccionar UN solo elemento en particular.

Clases: podemos asignarle el nombre de una clase a un grupo de elementos html.

Selectores básicos

- **Selector de tipo**: Selecciona todos los elementos que coinciden con el nombre del elemento especificado. Sintaxis: eltname Ejemplo: input se aplicará a cualquier elemento `<input>`.
- **Selector de clase**: Selecciona todos los elementos que tienen el atributo de class especificado. Sintaxis: .classname Ejemplo: .index seleccionará cualquier elemento que tenga la clase "index".
- **Selector de ID** Selecciona un elemento basándose en el valor de su atributo id. Solo puede haber un elemento con un determinado ID dentro de un documento. Sintaxis: #idname Ejemplo: #toc se aplicará a cualquier elemento que tenga el ID "toc".
- **Selector universal** Selecciona todos los elementos. Opcionalmente, puede estar restringido a un espacio de nombre específico o a todos los espacios de nombres. Sintaxis: * ns|* / Ejemplo: * se aplicará a todos los elementos del documento.
- **Selector de atributo** Selecciona elementos basándose en el valor de un determinado atributo. Sintaxis: [attr] [attr=value] [attr~=value] [attr|=value] [attr^=value]

[attr$=value] [attr*=value] Ejemplo: [autoplay] seleccionará todos los elementos que tengan el atributo "autoplay" establecido (a cualquier valor).

Anatomía de las reglas de estilo

Ahora que sabemos como *seleccionar* los elementos a los que queremos aplicar las reglas podemos escribir qué reglas queremos que se apliquen. Para el ejemplo vamos a usar la etiqueta `<style\/>`.

```
<style>    body {}    .divClass {}    #divId {}  </style>
```

En el ejemplo de arriba vemos tres selectores. El primero es para el elemento `body`, el segundo para todos los elementos de la clase `divClass` y el tercero para el elemento que tenga el id: `divId`. Dentro de los `{}` vamos a escribir todas las reglas que queremos que se apliquen a esos elementos.

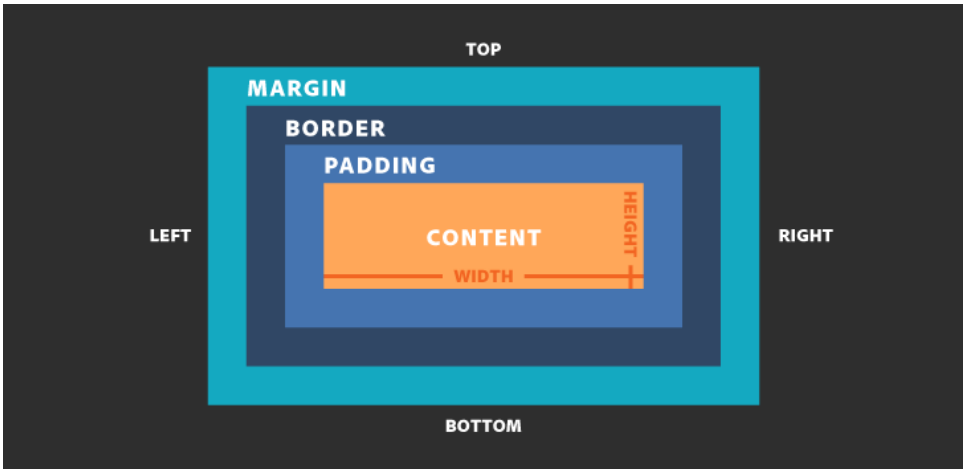
Aplicando reglas

Ahora que tenemos los elementos seleccionados podemos empezar a agregar las reglas que habíamos visto antes.

```
div {    propiedad: valor;  }
```

Introducción al modelo de caja (Box Model)

Para poder entender y luego manipular la forma en que los elementos HTML aparecen distribuidos en la página, tenemos que aprender cómo son representados estos en el browser.



En un documento html cada elemento es representado como una *caja rectangular* y en CSS cada una de estas cajas tiene 4 capas que podemos manipular. Yendo desde afuera hacia adentro, las capas son:

- **margin**: el espacio que separa al elemento de los otros elementos. Si los pensamos como cajas, es el espacio entre las cajas.
- **border**: el "borde de la caja". Podemos hacerlo visible con diferentes grosores, estilos y colores, como ya hicimos varias veces en ejercicios anteriores.

- **padding**: el espacio entre el borde de la caja y su contenido. En la metáfora de la caja, podríamos por ejemplo tener una caja grande con algo chiquito adentro, osea que "habría mucho padding".
- **content**: el contenido de la caja. Por ejemplo el texto en un h1, otros tags anidados, etc, todo lo que esté contenido en el elemento.

height (alto) y width (ancho)

Podemos decirle al navegador exactamente qué tan *ancho* y *alto* queremos que sea nuestro elemento (contenido), esto se usa en divs, imgs y otros elementos basados en la altura (para determinar el tamaño del texto, necesitaremos usar un propiedad de estilo diferente). Los valores de tamaño pueden estar en muchas medidas diferentes, pero el más común es el píxel "px".

```
div {    height: 400px;    width: 400px;  }
```

margin

El margen es el área transparente alrededor del elemento que deseas que no choque con nada. Es la capa más externa en el Modelo de caja.

border

Borde establecerá un *borde* alrededor de su elemento, puedes determinar el tamaño, color y estilo del borde. Puede encontrar una lista de estilos de borde aquí: https://developer.mozilla.org/en-US/docs/Web/CSS/border. El borde está fuera del padding, pero dentro del margen.

```
div {    border: 1px solid black;  }
```

padding

El padding es el area transparente entre el borde y el contenido, es similar al margen, pero para adentro

Cálculo del modelo de caja

Cuando establecemos el *alto* y el *ancho* de un elemento a traves de la regla css *height* y *width*, sólo estamos configurando el contenido. Para calcular la altura y el ancho reales, tenemos que tener en cuenta el padding, el borde y el margen.

- El padding es un área transparente alrededor del contenido.
- El borde se envolverá alrededor del relleno
- El margen es el área transparente más externa que envuelve toda la caja.

Por ejemplo. Si establecemos la altura del contenido en 20 px y el ancho en 20 px, el relleno en 5 px, el borde en 1 px y el margen en 10 px.

Altura real = 25px (contenido) + 2 * 5px (relleno, cada lado) + 2 * 1 (borde de cada lado) + 2 * 10 (margen, cada lado) = 57px

Ancho real = 25px (contenido) + 2 * 5px (relleno, cada lado) + 2 * 1 (borde de cada lado) + 2 * 10 (margen, cada lado) = 57px

Saber esto nos ayudará a dimensionar y posicionar nuestros elementos correctamente.

Un par de otras propiedades CSS

background

El background se puede establecer en una variedad de reglas, la más común sería establecer el fondo en un color o una imagen. Ambos se muestran a continuación.

```
 .divClass {    background: red;  }  #divId {    background: url
('http://imageurl.com/image.jpg');  }
```

color

El color se usa sólo para texto. Establecerá el color de tu texto

font-size

No podemos usar ancho o alto para el texto, pero podemos determinar el tamaño de la fuente utilizada. Puede usar cualquier unidad de tamaño aquí que usaría con una fuente en un procesador de textos (px, em, in, etc.). El más popular es px.

**Hojas de estilo externas y el elemento **

Hemos explicado cómo usar el elemento html \ `<style>`. Esto está bien si tiene una página web muy pequeña y un estilo mínimo, pero la mayoría de las páginas comenzarían a sentirse abarrotadas muy rápidamente si incluimos todo nuestro CSS en el HTML. Afortunadamente, tenemos una solución para eso, hojas de estilo externas y el elemento \ `<link>`.

Una hoja de estilo externa es simplemente otro archivo con el tipo de archivo .css. Convencionalmente, este archivo se llama algo así como "style.css". Podemos tomar todas las reglas de estilo que escribimos entre las etiquetas \ `<style>` y transferirlas directamente al archivo css. No necesitamos incluir nada más, solo las reglas de estilo.

Una vez que tengamos una hoja de estilo externa creada, necesitaremos asegurarnos de que el navegador lea ese archivo y aplique las reglas a nuestra página. Le decimos al navegador que busque ese archivo utilizando el elemento \ `<link>`. Podemos eliminar las etiquetas \ `<style>` y en su lugar agregar el elemento \ `<link>`. Dentro del elemento de enlace, necesitaremos proporcionar la ubicación y el tipo de archivo que estamos vinculando. Utilizaremos dos banderas, la bandera "rel" y la bandera "href".

La bandera rel solo le dirá al navegador qué tipo de archivo es y cómo procesarlo. En nuestro caso lo configuraremos como "hoja de estilo"

La bandera href le dirá al navegador dónde encontrar el archivo. Si el archivo está en la misma carpeta que nuestro archivo html, podemos configurarlo en: "./styles.css" (esta ruta será relativa)

```
 <link rel = "stylesheet" href = "./ styles.css" />
```

Ahora que tenemos nuestra hoja de estilo externa vinculada a nuestro archivo HTML, deberíamos ver las reglas de estilo que establecemos reflejadas en nuestra página.

Lección 9: CSS Intermedio

En esta lección cubriremos:

- Introducción al posicionamiento.
- Propiedad "display".
- Propiedad "position".
- Usar "position" para posicionar elementos.
- Introducción a Flexbox.

<iframe src="https://player.vimeo.com/video/425169846" width="640" height="564" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Introducción al posicionamiento

Armar un layout y hacer que todo se vea limpio es lo que la mayoría de gente espera hacer cuando empiezan a aprender CSS. Posicionar elementos HTML en la página con CSS es posiblemente la habilidad más poderosa que tiene CSS, aunque también puede ser la más **frustrante**. En esta lección aprenderemos distintas formas de posicionar elementos en la página.

La propiedad "display"

Esta propiedad es una de las más importante de posicionamiento en CSS. Podemos usarla para controlar cómo se muestra el contenido en relación a los elementos alrededor de este, y cómo se comportan en la pantalla.

```
div {    display: <valor de la propiedad>;  }
```

Hay dos tipos de elementos "display" ya incluidos en HTML; "block" e "inline",

Elementos "block"

Un elemento "block" siempre arrancará en una nueva línea, y siempre tomará el ancho máximo del contenedor en el cual se encuentre. ¿Recuerdas que en la lección anterior aprendimos que el elemento `<p>` siempre empieza en una nueva línea? Es porque es un elemento "block", como también lo son los `div` y los `<h1-6>`.

Elementos "inline"

Los elementos "inline" son opuestos a los "block", dado que no comenzarán en una nueva linea y sólo tomarán el espacio suficiente que necesiten para mostrar la información dentro del mismo. Los elementos `<span>`, `<a>` e `<img>` son inline.

Podemos controlar cómo un elemento se comporta usando la propiedad "display". Si queremos que un elemento "inline" actúe como uno "block", le definimos:

```
div {    display: block;  }
```

Y vice-versa.

"Display" tiene otras reglas que debemos conocer:

none

Definir como valor "none" hará que el elemento desaparezca completamente. No debe ser confundido con la regla `visibility: hidden;` dado que esta hace que el elemento sea invisible pero no lo elimina de la página (el espacio continuará ocupado por *algo*). Si un elemento está definido como `display: none;` no habrá signos de él en la página.

flex

Flex es una nueva herramienta que nos ofrece CSS3, la cual nos da la habilidad de controlar en qué parte de la página queremos que estén nuestros items. Hablaremos sobre ella más adelante en esta lección.

grid

Esta es una característica de CSS3 que nos permitirá crear sistemas de grillas dentro de un elemento.

La propiedad "position"

Esta propiedad especificará qué tipo de método de posicionamiento usará un elemento HTML. Hay 5 métodos diferentes disponibles (aquí veremos 4 de ellos).

```
div {    position: <valor de la propiedad>;  }
```

static

Este es el posicionamiento por defecto de un elemento, definir un elemento como "estático" no afectará al comportamiento del mismo de ninguna manera.

relative

Usar este valor mantendrá el elemento posicionado como si fuese estático, pero nos permitirá usar otros métodos de posicionamiento de elementos que veremos enseguida.

fixed

Definir un elemento como "fijo" hará que éste quede fijo en un lugar de la pantalla, sin importar cuánto navegemos, "scroleemos" o movamos la pantalla, el elemento se quedará en ese lugar. Los casos de uso pueden ser un header o la barra de navegació de un sitio web.

absolute

"absolute" es muy parecido a "fixed", excepto que el elemento quedará anclado de forma relativa al elemento *parent* (siempre que el elemento padre tenga alguna posición definida, excepto "static").

Usar propiedades de posición para posicionar elementos

Ahora que hemos definido nuestros métodos de posicionamiento al estilo que queremos usar, podemos arrancar a posicionar nuestro elemento. (Nota: esto funciona para cada método de posición que no sea "static", dado que no afecta al elemento de ninguna forma).

top, left, right y bottom

Después de haber definido nuestro método de posicionamiento, podemos usar las propiedades "top", "left", "right" y "bottom" para acomodar nuestro elemento. El valor que le des a cada uno determinará qué tan lejos del borde quedará nuestro elemento. Por ejemplo, si queremos que nuestro elemento esté en la esquina superior izquierda (con una posición fija), podemos usar lo siguiente:

```
div {    position: fixed;    top: 0;    left: 0;  }
```

Si lo quisiésemos 10 píxeles debajo del límite superior y 10 píxeles del borde derecho:

```
div {    position: fixed;    top: 10px;    right: 10px;  }
```

Introducción a Flexbox

Introducido en CSS3, Flexbos es una nueva e interesante característica. La misma nos permite posicionar nuestros elementos en relación a su *parent* y entre ellos. Ya no necesitamos aplicar "hacks" para cosas como centrar elementos. Esto nos permite que el diseño "mobile-friendly" sea excelente y nos hace dedicar menos tiempo tratando de posicionar elementos como corresponde. Flexbox se puede complicar muy rápido, pero veremos los aspectos básicos a continuación.

"display: flex" e "inline-flex"

Como mencionamos anteriormente en la sección de la propiedad "display", uno de los valores puede ser "flex", esto hace que cualquier contenedor sea un "flex block". También podemos usar "inline-flex" para hacer que el elemento sea "flex" e "inline", aunque para la mayor parte del tiempo, usaremos simplemente "flex".

"justify-content" y "align-items"

Ahora que nuestro contenedor (elemento) es "flex", podemos imaginarlo como una grilla con columnas que van de izquierdo a derecha y filas que van de arriba a abajo. Podemos usar las propiedades "justify-content" y "align-items" para decirle al contenedor dónde queremos que estén los elementos en la grilla. En principio, "justify-content" aplicará al movimiento de izquierda a derecha (fila), y "align-items" lo hará de arriba a abajo (columna). Tenemos unas reglas que debemos aplicar a cada una de estas reglas:

- `center`: centrará el elemento (o grupo de elementos) a lo largo de un eje en el que aplica esta regla.
- `flex-start`: Este es el valor por defecto de cada "flex box", mostrará todos los elementos en un grupo al comienzo de una fila o columna.
- `flex-end`: es lo opuesto a `flex-start`, mostrará los elementos al final de un grupo al comienzo de una fila o columna.
- `space-between`: Esta regla espaciará uniformemente el elemento o los elementos a lo largo de la fila o columna. El primer elemento estará como `flex-start` y el último como `flex-end`.

- `space-around`: Similar a `space-between`, pero aplicará márgenes igualitarios entre cada elemento, por lo que ningún elemento estara directamente sobre el borde del contenedor.

Ejemplo: si quisiésemos nuestros elementos centrados en el medio exacto de un "flex box", usaríamos lo siguiente:

```
div {    display: flex;    justify-content: center;    align-items: center;  }
```

"flex-direction"
Esta propiedad puede cambiar cómo el navegador interpreta `justify-content` (JC) y `align-items` (AI). El valor por defecto es "row" (fila), y esto funciona en la mayoría de los casos, pero algunas veces queremos cambiar cómo funciona la dirección del contenido.

- `row`: es la dirección por defecto. JC aplica de izquierda a derecha, AI aplica de arriba a abajo.
- `column`: Esto invertirá qué propiedad controla qué dirección. JC aplicará de arriba a abajo y AI de izquierda a derecha.
- `row-reverse`: Sólo invierte la dirección de JC de derecha a izquierda, no afecta a AI.
- `column-reverse`: Sólo invierte la dirección de AI de abajo a arriba, no afecta a JC.

"align-self"
Por último, cubriremos una propiedad más avanzada llamada "align-self". La misma será aplicada a un elemento dentro del "flex box" del cual queremos separar el control de `align-items`. Si le damos la propiedad `align-self`, podemos colocarla en cualquier lugar a lo largo del eje de elementos de alineación que queramos. (Nota: NO existe `justify-self`, esta es la razón principal por la que los desarrolladores cambiarán la dirección de "flex").

Expresiones vs Statements
<iframe src="https://player.vimeo.com/video/480856050" width="640" height="360" frameborder="0" allow="autoplay; fullscreen" allowfullscreen></iframe>

Podemos decir que todo el código que escribimos en JS o "hace algo" o "retorna algo" (o una combinación de los dos). En la terminología de lenguajes de programación esta diferencia está clasificada en la definición de **expressions** (expresiones) y **statements** (sentencias).

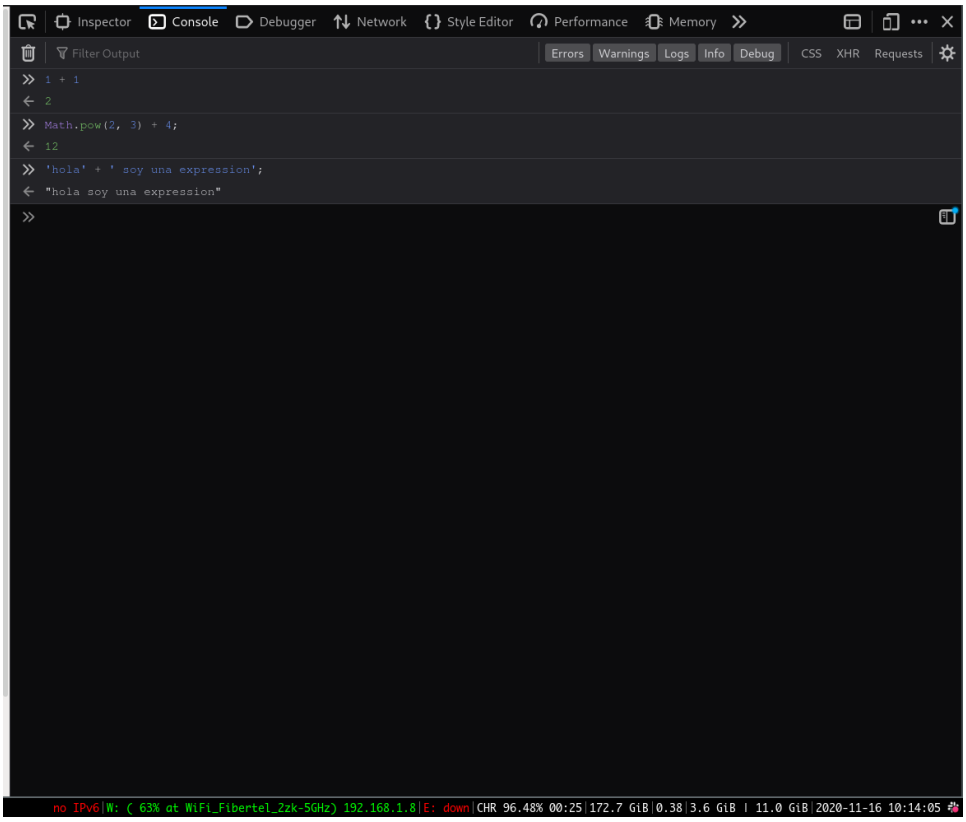
Podriamos definir conceptualmente a ambas como:

- Una **expression** siempre se convierte (retorna) un valor.
- Un **statement** realiza una acción.

Cuando escribimos código, todo el tiempo mezclamos expresiones y statements para conseguir el resultado final. Por lo tanto, al principio es un poco díficil ver la diferencia entre las dos, pero vamos a intentar ejemplificar lo anterior:

```
// retorna algo1 + 1Math.pow(2, 3) + 4;'hola' + ' soy una expression';
```

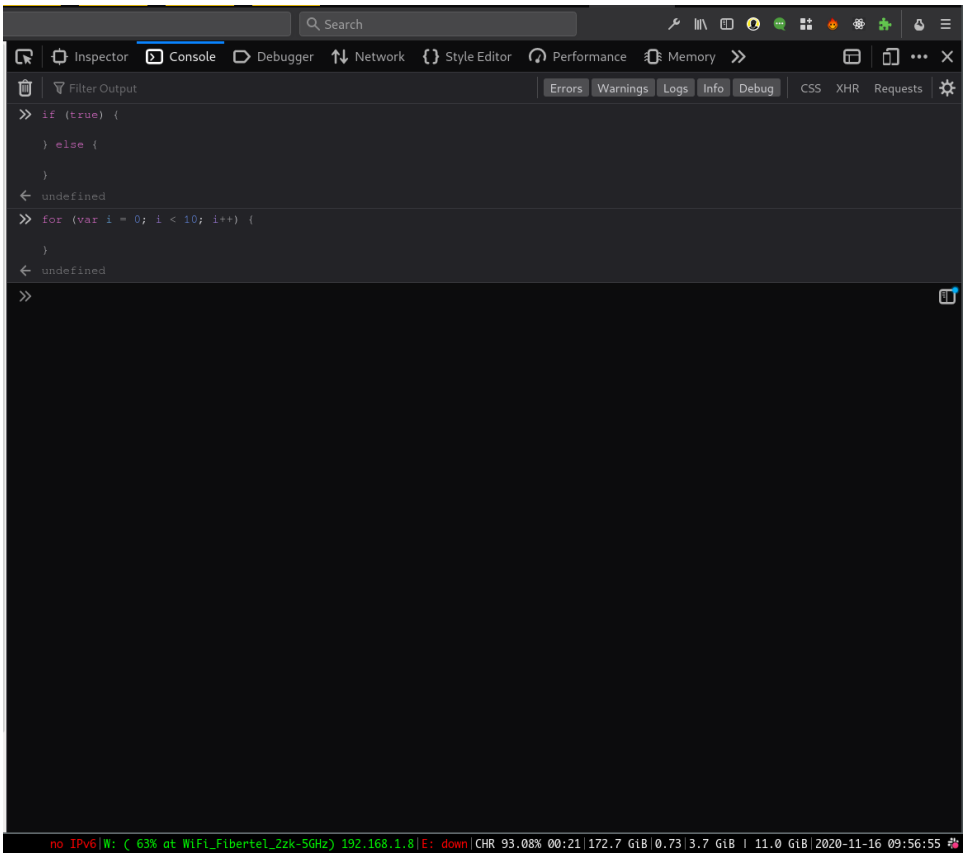
`1 + 1` intuitivamente se convierte o resuelve a `2`! eso es una expresión. Es cualquier cosa que escribamos y esperamos que se convierta en otro valor. Cuando *pegamos* una expression en la consola de Firefox o de Chrome, vamos a poder ver el resultado al que resuelve:




```
// hace algoif (condicion) {  // código ejecutado si es true} else {  // código ejecutado si es false}
```

En este ejemplo, vemos que el `if` "hace algo", es decir, escribimos el if para que bifurque le ejecución del código según el valor de `condicion`. Como ya sabemos, si `condicion` tiene un valor *verdaderoso* entonces se ejecutará el bloque de código de arriba, y si no, el de abajo. Fijensé que acá es importante discriminar lo que está *adentro* de los bloques de código, ya que ahí adentro podriamos escribir **expresiones** que sí devuelvan algo.

Nos podemos dar cuenta que algo es un statement, porque si lo *pegamos* en la consola del intérprete -por ejemplo, en la consola del Firefox o Chrome- vamos a ver que no produce ningún resultado:



Una regla fácil para distinguir entre una *expression* y un *statement* en JS es la siguiente: Si podemos ponerlo dentro de un `console.log`, es una *expression*, si no, es un *statement*. Por ejemplo:

```
 // expresiones!  console.log(1 + 1);  console.log(Math.pow(2,3) + 22);    // statements  console.log(if( true) {    // código  });  // jamás haríamos esto de arriba, no?
```

El **operador ternario**, es una expresión o un statement? ej: (numero > 10 ? 'mayor' : 'menor');

Expressions

Cómo dijimos arriba, una *expression* es cualquier pedazo de código **que pueda ser evaluado a un valor**. Justamente por esto, las vamos a usar en lugares donde JavaScript *espera un valor*. Por ejemplo, cómo cuando pasamos una expresión como argumento de una función.

Según la documentación de MDN, las expresiones se pueden clasificar en las siguientes categorías:

Expresiones Aritméticas

Son las expresiones que resuelven a un valor **numérico**. Por ejemplo:

```
10;1 + 10;2 * 16;
```

Expresiones de Strings

Son expresiones que resuelven a una **string**. Por ejemplo:

```
'hola';'hola' + ' como va?';
```

Expresiones lógicas

Son expresiones que resuelven a un valor **booleano**. Por ejemplo:

```
10 > 9;11 === 2;false;
```

Expresiones primarias

Son expresiones que se escriben por si mismas, y no utilizan ningún operador. Incluyen a valores literales, uso de variables, y algunos keywords de JS. Por ejemplo:

```
'hola';23;true;this;  // hace referencia al keyword thisnumero; // hace referencia a la variable numero
```

Expresiones de asignación

Cuando utilizamos el operador `=` hablamos de un *assigment expression*. Está expresión retorna el valor asignado. Por ejemplo:

```
a = 1; // si probamos esto en la consola, vemos que retorna el valor 1.var c = (a = 2); // vamos a ver que dentro de la variable c, está el valor retornado por la expresion `a = 2`
```

Este es un caso muy particular, nótese que esta expresion retornar una valor, **pero a su vez hace algo**!! Ese algo, es guardar el valor a la derecha del signo `=` en la variable a la izquierda del signo `=`. Otra cosa a notar, es que si usamos el keyword `var` la expresión retorna `undefined`, es decir, no es lo mismo una asignación que una declaración de variables.

Expresiones con efectos secundarios (side effects)

Son expresiones que al ser evaluadas retornan algo, pero a su vez tienen *un efecto secundario* (incrementar un valor, etc...). Por ejemplo:

```
contador++; // retorna el valor de contador e incrementa uno.++contador; // incrementa el valor de contador y retorna el valor;mult *= 2;// multiplica mult por dos, asigna ese valor a mult y retorna el valor;
```

Statements (sentencias)

Los *Statements* son instrucciones que le damos al intérprete de JS para que **haga algo**, ese algo puede ser: crear una variable, ejecutar un bloque de código N veces, ejecutar un bloque de código o no según una condición de verdad, declarar una función, etc...

Podemos clasificar a los Statements en las siguientes categorías:

Declaration Statements:

Este tipo de statements indican al intérprete que declare variables o funciones, se utiliza el keyword `function` y `var`. Por ejemplo:

```
var prueba; // declaro la variable pruebavar toni;  // declaro la variable
tonifunction suma(a, b) {  // declaro la función suma;  // bloque de código}
```

Habiamos dicho que por regla general lo que podamos pasarle a una función (por ejemplo, `console.log`) por argumento era una expresión... y muchas veces pasamos una declaración de una función por argumento. Esto sucede porque en JS existen tambien las **function expressions**.

Function expressions vs function declarations

Cuando declaramos una función el intérprete puede *interpretarla* como un statement o cómo una expresión, dependiendo del contexto. Por ejemplo:

```
//function declarationfunction resta(a, b) { // bloque de código}// function
expressionvar resta = function (a, b) {  // bloque de código}array.map(function() {
// código;});// el argumento de la función espera una expression// Immediately
Invoked Function Expression(function () {    console.log('IIFE');})();
```

Cómo vemos en el ejemplo de arriba, el intérprete *hace algo*: declara la función. Por lo tanto es un statement. En cambio, en el segundo ejemplo, estamos haciendo una asignación, y la asignación espera una *expresión* en la parte de la derecha, asi que le estamos pasando un function expression.

Nótese que un function expression puede no tener nombre. Estas son las llamadas **funciones anónimas**.

Conditional Statements:

Estos statements sirven para controlar el flujo de ejecución de código según si se cumple o no una condición. Por ejemplo:

```
if (condicion) { // condicion puede ser cualquier expression!!  // ejecuta este
bloque si condicion es true} else if (condicion2) {  // ejecuta este bloque de código
si condicion no es true y condicion2 es true} else {  // ejecuta este bloque de
código si condicion y condicion2 no son true.}
```

Loops (bucles) y Jumps (saltos)

Estos statements también controlan el flujo de ejecución del código, pero hacen que un bloque se ejecute N veces (ej: `for`), o que la ejecución salte a otro contexto (ej: `return`). Por ejemplo:

```
// loopswhile(condicion) { // condicion es una expresión!!  // ejecuta este código
mientras condicion sea true;}for (var i = 1; i < 10; i++) {  // ejecuta este bloque
de código 9 veces;}// jumpsfunction () {  // bloque de código  return;  // cuando
llegue acá, sale de la ejecución de la función y retorna un valor;  // bloque de
código}for (var i = 1; i < 10; i++) {  // ejecuta este bloque de código N veces;
continue; // salta a la siguiente iteración del bucle;  // desde acá no se
ejecuta;}throw new Error('hubo un error, se termina la ejecución');
```

Expression Statements

JS tiene la particularidad qué en donde sea que el intérprete espera un *statement*, nosotros podemos pasarle una *expresión*. Esto da lugar a los llamados *expression statements*.

Esto no funciona en sentido inverso, donde se espera una expresión *NO* podemos pasar una statement.

Proceso de subida de homeworks

IMPORTANTE: Luego de completar cada una de las homeworks del día deberán ejecutar el siguiente comando para subir sus trabajos a sus repositorios (Deben estar posicionados sobre la carpeta principal `Curso.Prep.Henry` para que funcione y haber corrido previamente `npm install`:

Para la homework número 1 no es necesario ya que su trabajo se realiza por fuera de este repositorio

```
node submit.js {numeroHomework}
```

En donde dice {numeroHomework} deben reemplazarlo simplemente por el número de homework correspondiente, por ejemplo `node submit.js 2`