PREP Henry

# CLI

La **interfaz de línea de comandos o CLI** , en inglés, command line interfaces, que no es más que una **caja negra** que puede recibir comandos a través de la escritura por medio de un teclado conectado a la computadora, procesarlos y arrojar un resultado, ya Sea positivo si el comando se leyó con éxito, o negativo si escribimos algo erróneo o la computadora no pudo procesar el comando escrito.

## Comando

Un **comando** es simplemente una **instrucción** que al ejecutarlo en la CLI realiza una **acción**, cuando se recibe el comando el sistema operativo ejecuta un proceso que puede leerlo, manipular datos y producir una salida.

**Comandos más importantes para el manejo de la terminal:**

* **pwd** : imprimir el directorio en donde nos encontramos.
* **ls** : imprime archivos o carpetas dentro del cual estamos trabajando.
* **mkdir** carpetaPrueba: crea una carpeta llamada ‘carpetaPrueba’ en el directorio que estamos situados. Si queremos crear una carpeta que lleve espacios en su nombre se debe utilizar comillas. Por ejemplo “PrepCourse Henry”
* **cd** : para ir hacia el directorio que contiene la carpeta o directorio en donde nos encontramos. También podemos usarla para movernos entre carpetas, por ejemplo al usar  **cd carpetaPrueba**  cambiamos de la carpeta actual hacia “carpetaPrueba”
* **touch** archivo.txt : crea un archivo con nombre “archivo.txt”
* **rm** archivo.txt : elimina el archivo “archivo.txt”
* **rm -r** carpetaPrueba: elimina la carpeta de nombre “carpetaPrueba”

“Recuerden escribir los comandos en **minúsculas** para que funcionen correctamente”.

## Token

**ghp\_iEROOWx5An1B4lVMCT3VWTZ9kdWFnC3RGfb8**

## Repositorios

Terminando que un repositorio no es más que un almacén o lugar donde se guardan ciertas cosas y que hoy en día esto gracias a la tecnología se da mayoritariamente en la nube, se creo GIT. Este sistema nos permite...

* Guardar, mantener y manipular archivos a lo largo del tiempo
* Posibilidad de trabajo colaborativo
* Proyectos guardados en un almacén “repositorio”

### Recordemos la información clave de la clase de video...

Un commit no es más que una captura instantánea de los cambios previstos en ese momento del proyecto. Las instantáneas confirmadas pueden preferir como versiones “seguras” de un proyecto.

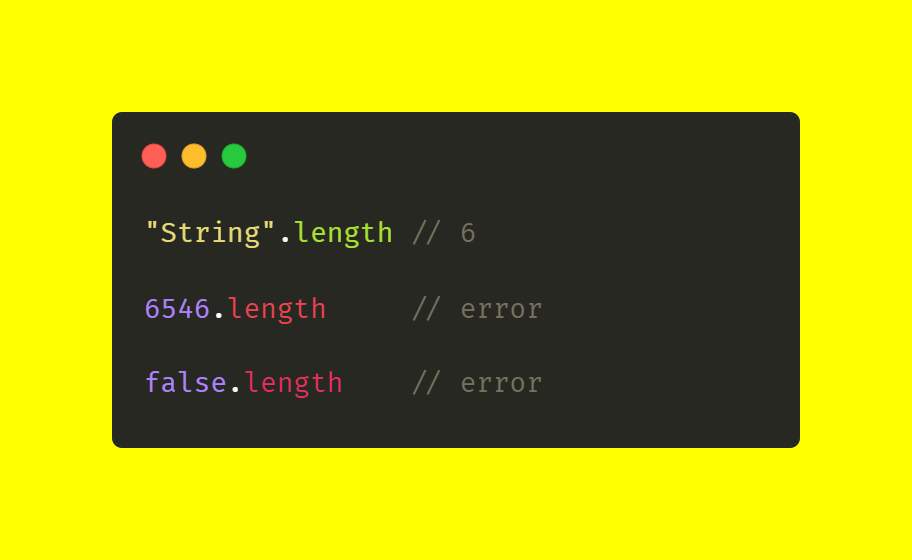
A continuación de compartimos los comandos más usados ​​con git:

* git init : comando para inicializar un repositorio local.
* git add: comando para que nuestro repositorio sepa de la existencia de un archivo o de sus últimos cambios, no almacene las actualizaciones de forma definitiva, únicamente las guarda en algo que conocemos como “Staging Area” (área de montaje o ensayo). Se puede armar como git add ArchivoEjemplo.js (solo agrega ese archivo) o git add (agrega todos los archivos modificados de la carpeta donde estas con el punto.
* git commit -m “ mensaje “ : comando para almacenar definitivamente todos los cambios que por ahora viven en el área de ensayo. En el mensaje ponemos el mensaje que explica los cambios cometidos.
* git push : comando para mandar nuestros commits a un servidor remoto.
* git status : ofrece una descripción del estado de los archivos.
* git pull : sirve para recibir cambios de repositorio remoto a local.
* El comando git log es una herramienta básica de Git para explorar el historial del repositorio. Este comando se usa cuando necesitas buscar una versión concreta de un proyecto o saber los cambios que se introducirán mediante la fusión en una rama de función.

Recuerda que Git nunca cambiará las versiones seguras de tu proyecto, a no ser que se lo pida expresamente. De esta forma, estamos guardando el estado actual de nuestro proyecto, para volver a visitarlo de ser necesario y por supuesto para mantenerlo a lo largo del tiempo.

#### MÉTODO LENGTH

Este es un método que solo se puede aplicar al tipo de dato string (más adelante veremos que también lo podemos utilizar en arreglos). En este ejemplo podemos ver que si lo intentamos aplicar en un número o en un booleano, nos devuelve error.



# Operadores Lógicos

Existen tres tipos de operadores lógicos: la negación o NOT , la conjunción o AND , y la disyunción u OR .

| **Operador** | **SIGNIFICADO** | **ejemplo** | **Resultado** |
| --- | --- | --- | --- |
| && | y | (5 < 2) **&&** (5 > 3) | FALSO |
| || | o | (5 < 2) **||**(5 >3) | verdadero |
| ! | no | **!**(5 <2) | verdadero |

* Las expresiones se evaluarán en orden, y la computadora omitirá cualquier expresión redundante. En una declaración && , si la primera expresión es falsa, la segunda expresión no se evaluará porque AMBAS expresiones deben ser verdaderas.
* Lo mismo para la declaración || . Si la primera expresión es verdadera, la segunda no se evaluará porque solo debe haber una declaración verdadera para cumplir con los requisitos del operador.
* Utiliza paréntesis. Como vimos en el segundo ejemplo de la negación! , utilizamos paréntesis para evaluar PRIMERO lo que estaba dentro de ellos, y luego aplicamos la negación. Podemos ajustar cualquier expresión entre paréntesis y se evaluará antes que toda la expresión en conjunto.

Existen 3 tipos de nomenclatura principal: **camelCase, PascalCase, snake\_case.**

A continuación te compartimos algunos recursos adicionales, que te serán útiles para ampliar ampliar tu conocimiento sobre este estema.

**Expresión Switch**  
La expresión **Switch**evalúa una condición. Compara su valor con una instancia case, y ejecuta declaraciones asociadas a ese case, así como las declaraciones en los case que siguen.

switch (expresión) {

case valor1:

//Declaraciones ejecutadas cuando el resultado de expresión coincide con el valor1

[break;]

case valor2:

//Declaraciones ejecutadas cuando el resultado de expresión coincide con el valor2

[break;]

...

switch (expr) {

case 'Naranjas':

console.log('El kilogramo de naranjas cuesta $0.59.');

break;

case 'Manzanas':

console.log('El kilogramo de manzanas cuesta $0.32.');

break;

case 'Platanos':

console.log('El kilogramo de platanos cuesta $0.48.');

break;

case 'Cerezas':

console.log('El kilogramo de cerezas cuesta $3.00.');

break;

case 'Mangos':

case 'Papayas':

console.log('El kilogramo de mangos y papayas cuesta $2.79.');

break;

default:

console.log('Lo lamentamos, por el momento no disponemos de ' + expr + '.');

}

console.log("¿Hay algo más que te quisiera consultar?");

**Bucle Do-While**  
El bucle**Do-While**(hacer mientras) ejecuta una sentencia especificada, hasta que la condición de comprobación se evalúa como falsa. La condición se evalúa después de ejecutar la sentencia, dando como resultado que la sentencia especificada se ejecute al menos una vez.

do {

i += 1;

document.write(i);

} while (i < 5);

let result = '';

let i = 0;

do {

i = i + 1;

result = result + i;

} while (i < 5);

console.log(result);

// Expected output: "12345"

**Declaración Continue**  
La declaración **continue** se utiliza dentro de los Bucles For. Nos permite omitir alguna de las iteraciones si se cumple una condición específica.

i = 0;

n = 0;

while (i < 5) {

i++;

if (i == 3)

continue;

n += i;

}

checkiandj:

while (i < 4) {

document.write(i + "<br>");

i += 1;

checkj:

while (j > 4) {

document.write(j + "<br>");

j -= 1;

if ((j % 2) == 0)

continue checkj;

document.write(j + " is odd.<br>");

}

document.write("i = " + i + "<br>");

document.write("j = " + j + "<br>");

}

**Break**  
La declaración **break** se utiliza dentro de los Bucles For. Nos permite "romper" o finalizar el bucle con antelación si se cumple una condición específica.

function comprobarBreak(x) {

var i = 0;

while (i < 6) {

if (i == 3)

break;

i++;

}

return i \* x;

}

ARREGLOS

1. El método **push** () añade uno o más elementos al final de un arreglo, y devuelve la nueva longitud del arreglo.
2. El método **pop** () elimina y devuelve el último elemento de un arreglo.
3. El método **unshift** () agrega uno o más elementos al inicio de un arreglo, y devuelve la nueva longitud del arreglo.
4. El método **shift** () elimina y devuelve el primer elemento de un arreglo.

MÉTODOS VARIOS

1. El método incluye () determina si un arreglo incluye o contiene un elemento específico. Devuelve verdadero o falso en cada caso.
2. El método every () determina si todos los elementos en un arreglo satisfacen una misma condición.
3. El método split () convierte una cadena en un arreglo, donde cada elemento contendrá una subcadena, dependiendo del parámetro divisor que indiquemos.
4. El método  join () convierte un arreglo en una cadena, uniendo todos los elementos de este en una misma cadena.

MÉTODOS DE RECORRIDO

1. El método  forEach () nos permite recorrer un arreglo, realizando alguna acción en para cada elemento.
2. El método map () también nos permite recorrer un arreglo y realizar una acción por cada elemento. La diferencia es que este método devuelve un nuevo arreglo los elementos modificados.

**¿QUÉ OTROS MÉTODOS EXISTEN?**

1. En la lección anterior descubrimos características son algunos de los métodos más utilizados de los arreglos en JavaScript. Estos fueros: push , pop , shift , unshift , split , include , every , join , forEach  y map .
2. Pero, ¿sabías que existen aún más métodos? ¡Así es! Existen más de 40 métodos que puedes utilizar para realizar operaciones complejas de una manera mucho más sencilla. Te estarás preguntando qué tal son...
3. Existe una página web llamada Mozilla Developer  en la que podremos encontrar toda la documentación, información y ejemplos sobre JavaScript. En esta ocasión te compartimos [todos los métodos de arreglos en JavaScript](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array) [(se abre en una nueva pestaña)](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array). Podrás encontrar el listado en la columna izquierda de la página.

**Bucle For**

Crea un bucle que consiste en tres expresiones opcionales, encerradas en paréntesis y separadas por puntos y comas, seguidas de una sentencia ejecutada en un bucle.

**Sintaxis**: for ([expresión-inicial]; [condición]; [expresión-final]) sentencia

**Bucle While**

Crea un bucle que ejecuta una sentencia especificada mientras cierta condición se evalúe como verdadera. Dicha condición es evaluada antes de ejecutar la sentencia.

**Sintaxis**: while(condición)  
  sentencia

**Bucle infinito**

Esto no es un nuevo tipo de bucle, sino algo que se puede producir en los que ya aprendimos. Un bucle infinito se produce cuando un bucle comienza a ejecutarse, pero jamás termina. Esto se debe a que nunca se llega a la condición de quiebre.

**Esto consume muchos recursos de la computadora, por lo que hay que intentar evitarlos.**