**EXPRESSION AND STATEMENTS**

- Una \*\*expression\*\* siempre se convierte (retorna) un valor.

- Un \*\*statement\*\* realiza una acción.

**Expresiones:**

**-Artiméticas:** Resuelven un valor numérico

-**Strings:** Letras

-**Expresiones lógicas:** Resuelven un valor booleano (ejem: 0 > 9; 11 === 2;) Valores booleanos: Que pueden ser true o false

-**Expresiones primarias:** Son expresiones que se escriben por sí mismas, y no utilizan ningún operador. Incluyen a valores literales, uso de variables, y algunos keywords de JS

**-Expresiones de asignación:** Cuando tienen un ‘=’

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

```js

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;

```

**Function statement**: Requiere un nombre, no arroja un valor

**Function expression:** Espera una expresión como argumento, arroja un valor

**Sentencias (Statement)**

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

**Function expression vs Function declarations:**

Function resta (a,b){}= Function declaration: declara una función

Var resta= function (a,b){}= Function expression: Asigna a una variable una función. En este caso, la función puede no tener nombre y se llamaría **funciones anónimas**

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

if (condicion) { ..}

else if (condicion2) {}

else { }

**-Loop (bucles) y Jumps (saltos)**

While

For

Function

Return: Cuando llega acá, sale de la ejecución de la función y retorna un valor

Continue: Salta a la siguiente iteración del bucle

Throw new Error('hubo un error, se termina la ejecución');

**Expression statements:** Donde sea que el intérprete espere un statement, se le puede pasar una expresión. No funciona en sentido inverso. Donde se espera una expresión no se puede poner un statement

**Comandos del terminal:**

-date: Fecha

-ls: Lista de carpetas/archivos

-ls-la: Archivos con detalles y ocultos de las carpetas

-clear: Borra el output

-cd: permite moverse entre carpetas

-mkdir (Nombre directorio): crea una nueva carpeta en el directorio en el que se está trabajando

-rm (nombre archivo): remove (borra un archivo)

-rm-rf: Fuerza la eliminación de una carpeta

-head-n (número de líneas) (nombre del archivo): Muestra las n primeras líneas de un archivo

-touch (Nombre del archivo): para crear un nuevo archivo

**GIT Y GITHUB**

**GIT HUB:** Para compartir el código, encontrar otros proyectos o como portafolio

* **Botón CODE:** Sirve para poder obtener URL del REPO y poder clonarlo
* **Botón WATCH:** Para seguir el repositorio
* **Botón FORK:** Para hacer el challenge (Agregarla a mi repositorio)
* **Añadir colaboradores:** Repositorio/Settings/Collaborators:Añadir username o correo
* **Trabajo profesional con pull requests:** En un trabajo profesional generalmente se bloquea la rama del master y para enviar un código a dicha rama pasa por un **code review** y luego de su aprobación se unen códigos con los **merge request**. Para realizar pruebas enviamos el código a servidores que normalmente llamamos **Staging develop** (Servidores de pruebas). Luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación de estos, pasan al servidor de producción con el ya antes mencionado **merge request.**
* **Ignorar archivos del repositorio con .gitignore:** No todos los archivos que agregas a un proyecto deberían ir a un repositorio (contraseñas= con extensión .env, o cuando estás conectado a una base de datos).

**GIT:** Sistema de control de versiones, distribuido y open source. Permite recuperar versiones anteriores

**Estados de git:**

**-Untracked:** Desconocidos, solo en el discoduro, Git no sabe de ellos

**-Unmodified:** Que no se han modificado

**-Modified**: Que se les ha hecho una modificación

**-Unstaged**: Git sabe de ellos, pero no han sido agregados al repositorio por git add

**-Staged:** En el repositorio, pero no han sido guardados definitivamente porque falta git commit

**-Tracked**: ultima actulaización gracias a git commit, archivos que viven dentro de Git, no tienen cambios pendientes

Modified-> (Git add) Staged-> (Git commited) Tracked

**Git clone**: Se pega la URL que aparece en el botón CODE de github, pide las credenciales y permite clonar la carpeta desde github en el pc

**Git status**: Para verificar el status de Git

**Git add** (nombre del archivo o del cambio): Para añadir el cambio, no lo guarda de forma definitiva, lo guarda en staging area. Para convertirlos en staged

**Gitt add .** (Para agregar todos los documentos que aparecen en git status, pero únicamente del repositorio)

**Git commit -m** ‘’ para añadir el comentario del cambio. Vuelve los archivos del estado staged al estado tracked

**Git push**: Para enviar las modificaciones a Github (git push origin master)

**Git pull nombre del repositorio nombre de la rama**: Para traer al local las modificaciones de la web

**Git log**: Para ver los cambios del historial del archivo

**git reset/git Branch -d** para borrar un commit o una rama

**git reset HEAD**: nos ayuda a sacar los archivos del estado Staged para devolverlos a su estado anterior.

**git rm --cached**: Mueve los archivos que le indiquemos al estado Untracked.

**git rm --force**: Elimina los archivos de Git y del disco duro

**git show**: nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuando se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser más detallados.

**git diff**: nos muestra la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación.

**git checkout + Id del commit:** podemos volver a cualquier versión anterior de un archivo específico o incluso de nuestro proyecto entero. Esta es también es la forma de movernos entre ramas.

**git log:** Muestra la identificación de los comits

**git log --stat:** commit descriptivo con cantidad de líneas agregadas y removidas por archivo.

**Para volver al estado en el que el proyecto funcionaba**

**git reset --soft 'commit'**`: mantiene los archivos en el área de Staging para que podamos aplicar nuestros últimos cambios desde un commit anterior.

**git reset --hard 'commit'**`: borra toda la información que tengamos en el área de Staging. (Perdiendo todo para siempre).

**Git checkout master ‘archivo.txt’:** Volvemos a la version madre

**Git reset HEAD:** Saca los archivos del estado Staged para volverlos a su estado anterior

**Git commit -amend:** Cuando falta realizar cambios y hacemos commit, con este comando se agregan los nuevos cambios al commit anterior

**Para buscar atributos dentro del código:**

* git grep -c "atributo": Si queremos buscar cuántas veces utilizamos un atributo
* git grep -n "atributo": Indica en qué línea está lo que estamos buscando

**RAMAS/BRANCHES**

Rama **Master o Main:** Va a producción

Rama **Development:** Se alojan las nuevas características hasta que estén listas para unirse al main

Rama **Hotfix:** Errores que se solucionan aquí para unirse al main

**Unmerged:** archivos con conflicto por el comando merge

**git checkout master 'archivo.txt'`:** volvemos a la versión madre.

**git branch "nombre de rama" ó git checkout -b ‘nombre de la rama’** // crea una rama

**git checkout "nombre de la rama"** // cambiarse de rama

**git reset id-commit ó git checkout “rama”o “id-commit”:** para movernos en el tiempo a cualquier otro commit de cualquier otra rama. **SE DEBE HACER COMMIT ANTES DE CAMBIAR DE RAMA PARA NO PERDER LOS DATOS**

**git merge "cualquier otra rama":** Combina la rama actual con la que ponemos después del comando

g**it push (nombre del repositorio) (nombre de la rama):** Para publicar una rama local al repositorio remoto.

Para conectar el repositorio de GitHub con nuestro repositorio local

1. **git remote add origin <URL>** guarda un repositorio remoto con el nombre de origin
2. Verificar que la URL haya guardado correctamente con **git remote** o **git remote -v**
3. Se crea un repositorio remoto con **git init**
4. Proporcionar información personal con

git config – global user.name “tu nombre”

git config – global user.mail “[tu@email.com](mailto:tu@email.com)”

1. Se añade el repositorio remoto creado previamente en github con **git remoto add** **(nombre del repositorio) (URL)**
2. Traer el repositorio remoto y hacer merge para crear un commit con los archivos de ambas partes con:
   1. Git fetch
   2. Git merge
   3. Git pull (nombre del repositorio) -- allow-unrelated-histories
3. Con git push se guardan los cambios en el repositorio remoto y se unifican los archivos de ambas partes

**Merge:** Unión de dos ramas

**Crear una rama en el repositorio local:**

* Git Branch “Nombre de la rama”
* Git checkout -b “Nombre de la rama”
* Git push origin “nombre de la rama”: Para enviar la rama al repositorio remoto

**Servidores locales**

-Git lab

-Bitbucked

-Github

**Node:** Ejecuta códigos de java script

En los README.md se debe pegar bash

npm install

**INTRODUCCIÓN JAVASCRIP JSI**

REPL: READ EVALUATE PRINT LOOP

**Variables**

**-**var (nombre de la variable)= ‘Texto’; Variables que pueden ser pisadas

-const (nombre de la variable)= ‘Texto’; Variable que no puede ser pisada

**Tipos de datos**

**-**Strings

-Números

-Booleanos (True or false)

-Null: Vacío

-NaN: Non a Number

**Operaciones matematicas**

-%: Devuelve el restante de la division

**Funciones:** Agrupa varias operaciones

Function ‘Nombre función’ (argumentos) {código de la función}

-console.log (‘’) Agrupa un argumento y lo imprime

Const ‘Nombre variable’= Function

Alert(): Saca un letrero

Prompt(): Saca un letrero con espacio para escribir

Math. (): Conjunto de funciones matemáticas pre creadas

Math.round(): redondear

Math.ceil ()redondear hacia arriba

Math.floor () redondear hacia abajo

Number.isInteger() determina si es entero

Math.pow(numero, numero al que queremos elevarlo)

**Funciones return**

Function (){

Return ..

}

Devuelve una suma. Las funciones que no devuelven ningún dato, devuelven undefined. Las funciones solo devuelven un return, si hay algo debajo de este, no lo devuelve

**Condicionales**

If (condicional) {

Acción

} else {

Acción}

**Para comparar “==”**

**Para evaluar los homeworks: npm test ---. Cuando este bueno se tiene que agregar al repositorio: git add, git commit, pit push**

**JAVASCRIPT 2**

Todos los números son true, menos el cero

Todos los strings son true, menos vacio ‘’

Null, undefined son false

Si se le pone **!** se llama a la negació, **!!** (doble negación, para conocer el booleano en JS)

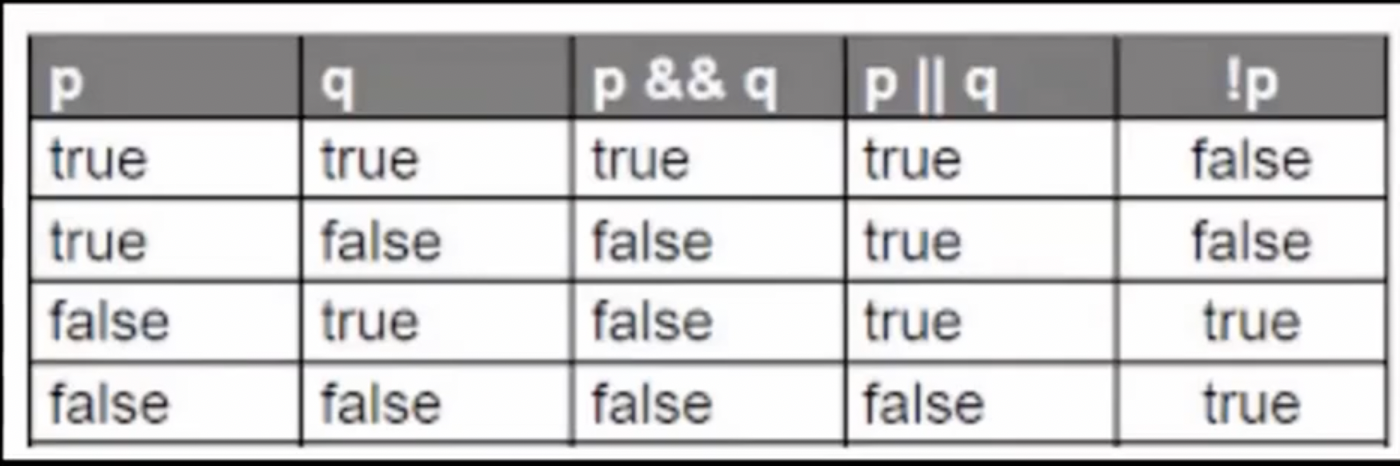
JS convierte números en strings o a booleano si lo requiere para no arrojar error.

**===** Te da la igualdad entre tipo de dato y valor

== Te da la igualdad del valor

!== Diferente

**Operadores lógicos (y/o):** Tabla de verdad

****

And: **&&**

O: **||**

No: **!**

**BUCLES**

**DRY CODE:** Dont repeat yourself

For (var i=0; i<=100; i++){

/Bloque de código

}

While (condicion){

Bloque de codigo

}

**Switch:** Para comparar una expression con un conjunto de statements

Switch(expr){

Case ‘comparacion1’:

Declaracion

Break;

ARREGLOS

Var array=[elementos separados por comas, puede ser cualquier tipo de datos]

[posición 0, posición 1, posición 2…]

Si en la posición x está una función, puedo llamarla array[x]() 🡪 aparece la función

Las strings internamente son arreglos

array.length: indica cuantos elementos tiene el arreglo

array[y].length: indica numero de elementos de la posición y del arreglo

**.push y .pop**

Arreglo.push(x): Pone una x en la ultima posición libre del arreglo

Arreglo.pop(x): Saca el ultimo argumento del arreglo, lo devuelve y lo elimina del arreglo

Arreglo.unshift: Agrega un elemento al inicio del arreglo

**JS IV: OBJETOS LITERALES**

Un objeto es una colección de propiedades: Una propiedad es una asociación entre un nombre (clave) y un valor (key (con lo que se identifica el value) value(resultado))

Var objeto={

Propiedad1: valor1,

…

PropiedadN: valorN,

}

Objeto[‘Propiedad’]=Valor

Si alguna de las propiedades es un valor para llamarla, objeto[‘propiedad]()

**Para llamar un objeto:**

-Bracket notation objeto[‘Propiedad’]

-DOT notation: objeto.propiedad (Es literal lo que se ponga aquí)

**Para eliminar una propiedad:**

Delete objeto.propiedad

Una función dentro de un objeto se conoce como método

**This:** this.propiedad pone la propiedad mencionada de un objeto (this hace referencia al objeto) dentro de una función. Permite usar una función en varios objetos

**CLASES JSV**

Una clase es una plantilla para la creación de objetos según un modelo predefinido. Se una para representar entidades o conceptos, como los sustantivos en el lenguaje.

**Function constr y new**

**Constr:** Función constructora que permite crear objetos (Crea la plantilla)

**New:** Crea objetos a partir de la función constructura

**Function** NombreDeLaClase(valor1, valor 2){

**this.**Propiedad1=valor1 || valor; //Si no se pone valor1, por defecto toma valor como propiedad 1

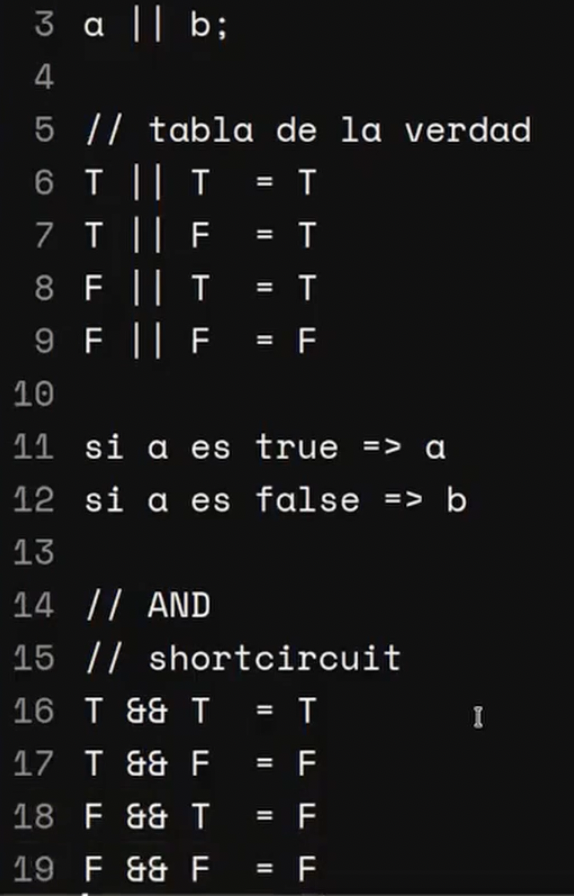
**this.**Propiedad2=valor2;

}

**var** objeto = **new** NombreDeLaClase(valor1, valor2, valor3)

console.log(objeto)

Todas los nombres de las funciones constructoras empiezan con mayúsculas



**Prototipo**

**Propiedades que tiene JS para realizar funciones determinadas. Todos los objetos tienen una referencia a otro objeto llamada proto**

Objetoaevaluar.\_\_proto\_\_ Arroja el nombre de la variable

**Function** NombreDeLaClase(valor1, valor 2){

**this.**Propiedad1=valor1 || valor; //Si no se pone valor1, por defecto toma valor como propiedad 1

**this.**Propiedad2=valor2;

}

NombreDeLaClase.prototype.getNombre=function{

Return this.valor1 + this.valor.2

}

**var** objeto = **new** NombreDeLaClase(valor1, valor2, valor3)

console.log(objeto)

**Object.create y Pure Prototypal Inheritance:** Object.create recibe un objeto como parámetro y crea un nuevo objeto cuyo prototipo es el primero que le pasamos. Por lo tanto, vamos a poder acceder a todos los métodos y propiedades del objeto base en los nuevos objetos creados

**var** person={

nombre=valor1;

apellido=valor2;

}

**var** Toni=**Object.create**(person) //Le psa el objeto person a Toni

Aparece como un objeto vacío porque todo está en el prototipo

**ES6 y clases (class)**

Las funciones se ponen en medio sin ningún carácter esp

**class** Persona {

**constructor**(Nombre, Apellido){

**this.**nombre=Nombre;

**this.**apellido=Apellido;

}

Saludar(){

Console.log(‘Hola’ + **this.**nombre)

}

**var** toni=**new** persona (‘Toni’, ‘Tralice’);

toni.saludar;

}

**Una clase puede heredar otra clase**

**Class** Empleado **extends** Persona {

**constructor** (nombre, apellido, empleo, sueldo){

**super**(nombre, apellido)//Invoca al constructor de la clase que extiendo

**this.**empleo=empleo;

**this.**sueldo=sueldo;

}

}

**var** toni= **new** Empleado (‘Toni’, ‘Reyes’, ‘Profesor’, ‘100’);

toni.saludar()