Cheat Sheet

**JavaScript**

**VARIABLES:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Var | Le indica a JavaScript que vamos a declarar una variable de tipo var a la cual le podemos asignar un valor. | var nombre = ‘Hackerman’; |
| Let | Solo será accesible en el bloque de código en el que fue declarada. | let contador = 0; |
| Const | Al igual que let solo es accesible en el bloque de código en el que fue declarada. Además, no podemos cambiar su valor. | const url = ‘http://digitalhouse.com. ar’; |

**TIPOS DE DATOS:**

|  |  |  |
| --- | --- | --- |
| **Tipos** | **Explicación** | **Ejemplo** |
| Numéricos | Pueden ser enteros o con decimales. | 5, 10, 20.5 |
| Strings | Cadenas de textos. Se escriben entre comillas dobles o simples. | “Hola Mundo” “tengo 5 manzanas” |
| Lógicos o Booleanos | Su valor puede ser true o false (verdadero o falso). | True or false |
| Objeto | Son colecciones de datos y en su interior pueden existir los anteriores. | {nombre: “Joaquín”, apellido: “Caggiano”, dni: 40826268} |
| Array | También son colecciones de datos y se escriben entre corchetes []. | [5, “Hola”, true] |
| NaN (not at number) | No es un número. |  |
| Null (nulo) | Los asignamos nosotros para indicar un valor vacío o desconocido. | Let pesoArgentino = null |
| Undefined | Las variables tienen un valor indefinido hasta que les asignamos un valor. | Let nombre |

**OPERADORES DE ASIGNACIÓN**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| = | Asignan el valor de la derecha en la variable de la izquierda | Let edad **=** 24 |

**OPERADORES ARITMÉTICOS:** nos permiten hacer operaciones matemáticas, devuelven el resultado de la operación.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **+** | Sumar | 35 **+** 2 |
| **-** | Restar | 4 **–** 3 |
| **\*** | Multiplicar | 5 **\*** 2 |
| **/** | Dividir | 10 **/** 5 |
| **++** | Incremento | 15**++** = 16 |
| **--** | Decremento | 15**--** = 14 |
| **%** | Módulo, te da el resto de la división | El resto de la división 15/5 es 0. |

**OPERADORES DE COMPARACIÓN SIMPLE:** comparan dos valores, devuelven true o false.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **==** | Igualdad | 10 == 15 (false) |
| != | Desigualdad | 10 != 15 (true= |

**OPERADORES DE COMPARACIÓN ESTRICTA:** comparan el valor y el tipo de dato también.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **===** | Igualdad estricta | 10 === “10” (false) |
| !== | Desigualdad estricta | 10 !== 15 (true) |

**OPERADORES DE COMPARACIÓN:** comparan dos valores, devuelven true or false.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| > | Mayor que | 15 > 15 (false) |
| >= | Mayor o igual que | 15 >= 15 (true) |
| < | Menor que | 10 < 15 (true) |
| <= | Menor o igual que | 10 <= 15 (true) |

**OPERADORES LÓGICOS:** permiten combinar valores booleanos, el resultado también devuelve un booleano

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| AND (**&&**) | Todos los valores deben evaluar como true para que el resultado sea true. | (10 > 15) && (10 != 20) false |
| OR (**||**) | Al menos un valor debe evaluar como true para que el resultado sea true | (10 > 15) && (10 != 20) true |
| NOT (**!**) | Niega la condición. Si era true, será false, y viceversa | ! false (true)  ! (20 > 15) (false) |

**OPERADORES DE CONCATENACIÓN:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **+** | Sirve para unir cadenas de texto. Devuelve otra cadena de texto. | Let nombre = “Joaquín”  Let apellido = “Caggiano”  Let nombreCompleto = nombre + “ “ + apellido |

**FUNCIONES:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **Function** *nombre de la función* (*parámetros*) {*código que queremos que se ejecute*} | Bloque de código que nos permite agrupar funcionalidad para usarla todas las veces que necesitemos. | Function sumar (a, b) {  return a + b  } |
| **Funciones declaradas** | Se declaran utilizando la estructura básica. | Function hacerHelado (cantidad) {  Return “conito”.repeat (cantidad)  } |
| **Funciones expresadas** | Se asignan como valor de una variable. | Let hacerSushi = function (cantidad) {  Return “sushi”.repeat (cantidad)  } |

**CONDICIONALES:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **if** (*condición*) {  *código a ejecutar si la condición es verdadera*  } | Nos permiten evaluar condiciones y realizar diferentes acciones según el resultado de esas evaluaciones. | Let edad = 19;  Let acceso = “ “;  If (edad < 16) {  Acceso = “Prohibido”;  } else if (edad >=16 && edad <=18) {  Acceso = “permitido con un acompañante mayor”  } else {  Acceso = “permitido”  } |
| **Else** | Agrega un bloque de código a ejecutar en caso de que la condición sea falsa. Es opcional. |
| **Else if** | Agrega un if adicional. Es decir, otra condición a evaluar en caso de que la primera sea falsa. |
| **If ternario**  *Declaramos una expresión que se evalue como true o false* **?** *si la condición es verdadera, se ejecuta este código* **:** *si la condición es falsa se ejecuta este código*. | Es obligatorio poner código en la segunda expresión. Si no queremos que pase nada, podemos usar un string vacío “ “. | 4 > 10 ? “El 4 es más grande” : “El 10 es más grande” |
| **Switch** (expresión) {  **Case** valorA :  //código a ejecutar si la expresión es igual al valorA  **Break**;  **Case** valorB :  // código a ejecutar si la expresión es igual al valorB  **Break**;  **Default**:  //código a ejecutar sin ningún caso es verdadero.  } | El switch está compuesto por una expresión a evaluar, seguida de diferentes casos, tantos como queramos, cada uno contemplando un escenario diferente. Los casos deben terminar con la palabra reservada **break** para evitar que se ejecute el próximo bloque.  Si queremos considerar que ninguno de los casos sea verdadero, utilizamos la palabra reservada **default** seguida de dos puntos **:** y el bloque de código que queremos que se ejecute. |  |

**ARRAYS:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Let miArray = **[***todos tipos de datos separados por comas***]** | Nos permiten generar una colección de datos ordenados. Usamos los corchetes para saber que es un array. Los elementos del array inician en el índice 0 hacia adelante. | Let miArray = [“Hola”, true, 24] |
| Longitud de un array: utilizamos **.length** | Nos sirve para saber la cantidad de elementos dentro de un array. | Let pelisFavoritas = [“Star Wars”, “Kill Bill”, “Alien”]  pelisFavoritas.length; *devuelve 3, el número de elementos del array*. |

**MÉTODOS DEL ARRAY:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **.push ()** | Agrega uno o varios elementos al final de un array.   * Recibe uno o más elementos como parámetros. * Retorna la nueva longitud del array. | Let colores = [“Rojo”, “Naranja”, “Azul”];  Colores**.push** (“Violeta”, “Amarillo”) |
| **.pop ()** | Elimina el último elemento de un array.   * No recibe parámetros. * Devuelve el elemento eliminado. | Let series = [“Mad Men”, “Breaking Bad”, “The Sopranos”];  Let ultimaSerie = series**.pop** ();  console.log (última serie) // [“The Sopranos”] |
| **.shift ()** | Elimina el primer elemento de un array.   * No recibe parámetros. * Devuelve el elemento eliminado. | Let nombres = [“Frida”, “Diego”, “Sofía”];  Let primerNombre = nombres**.shift ()**;  Console.log (primerNombre) // [“Frida”] |
| **.unshift ()** | Agrega uno o varios elementos al principio de un array.   * Recibe uno o más elementos como parámetros. * Retorna la nueva longitud del array. | Let marcas = [“Audi”];  Marcas.unshift (“Ford”);  Console.log (marcas) // [“Ford”, “Audi”] |
| **.join ()** | Une los elementos de un array utilizando el separador que le especifiquemos. Si no se lo especificamos, utiliza comas.   * Recibe un separador (string), es opcional. * Retorna un string con los elementos unidos. | Let dias = [“Lunes”, “Martes”, “Jueves”];  Let separadosPorComa = dias.join ();  Console.log (separadosPorComa); // “Lunes, Martes, Jueves”  Let separadosPorGuion = días.join (“ – “);  Console.log (separadosPorGuion) // “Lunes – Martes – Jueves” |
| **.indexOf ()** | Busca en el array el elemento que recibe como parámetro.   * Recibe un elemento a buscar en el array. * Retorna el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna **-1**. | Let frutas = [“Manzana” “Pera”, “Frutilla”];  Frutas.indexOf (“Frutilla”); //Devuelve el 2, el índice del elemento. |
| **.lastIndexOf ()** | Empieza a buscar el elemento por el final del array. | Let clubes = [“Racing”, “Boca”, “Lanús”, “Boca”];  clubes.lastIndexOf (“Boca”); // devuelve 3, el primer índice donde encontró el elemento. |
| **.includes()** | Similar a .indexOf(), con la salvedad que retorna un booleano.   * Recibe un elemento a buscar en el array. * Retorna *true* si encontró el elemento, *false* si no lo encontró. | Let frutas = [“Manza”, “Pera”, “Frutilla”];  Frutas.includes(“Frutilla”); //devuelve true. |

**JSON:** Javascript Object Notation. Es un formato de texto sencillo utilizado para el intercambio de datos entre distintos sistemas.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **JSON.parse ()** | Convierte un texto con formato JSON al tipo de dato equivalente de Javascript.   * Recibe una cadena de texto con formato JSON. * Devuelve el mismo dato que recibió en formato Javascript. | Let datosJson = “{“club”: “Independiente”, “barrio”: “Avellaneda”}”;  Let datosConvertidos = JSON.parse (datosJson);  Console.log (datosConvertidos); // se verá en consola un objeto literal. // {club: “Independiente”, barrio: “Avellaneda”} |
| **JSON.stringify ()** | Convierte un tipo de dato de Javascript en un texto formato JSON.   * Recibe un tipo de dato de javascript. * Devuelve una cadena de texto con formato JSON. | Let objetoLiteral = {nombre: “Carla”, país: “Argentina”};  Let datosConvertidos = JSON.stringify (objetoLiteral);  Console.log (datosConvertidos); //Se verán en consola los datos en un string de tipo JSON. // “{“nombre”: “Carla”, “país”: “Argentina”}” |

**MÉTODOS DE STRINGS:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **.length** | Retorna la cantidad total de caracteres del string, incluidos los espacios. | Let miSerie = “Mad Men”;  miSerie.length; //devuelve 7. |
| **.indexOf ()** | Busca, en el string, el string que recibe como parámetro.   * Recibe un elemento a buscar en el array. * Retorna el primer índice donde encontró lo que buscábamos. Si no lo encuentra, retorna **-1**. | Let saludo = “Hola! Estamos programando”;  Saludo.indexOf (“Estamos”); //devuelve 7  Saludo.indexOf (“vamos”); // no lo encuentra, devuelve -1.  Saludo.indexOf (“o”); //encuentra la letra “o” que está en la posición 2. |
| **.slice()** | Corta el string y devuelve una parte del string donde se aplica.   * Recibe 2 números como parámetros: el índice desde donde inicia el corte y el índice hasta donde hacer el corte. * Retorna la parte correspondiente al corte. | Let frase = “Breaking Bad Rules!”  Frase.slice (9, 12) // devuelve “Bad”. |
| **.trim ()** | Elimina los espacios que estén al principio y al final de un string.   * No recibe parámetros. * No quita los espacios del medio. | Let nombreCompleto = “ Homero Simpson “;  nombreCompleto.trim () // devuelve “Homero Simpson” |
| **.split ()** | Divide un string en partes.   * Recibe un string que usará como separador de las partes. * Devuelve un array con las partes del string. | Let canción = “And bingo was his name, oh!”;  Canción.split (“ “);  // devuelve [“And”, “bingo”, “was”, “his”, “name,”, “oh!”] |
| **.replace ()** | Reemplaza una parte del string por otra.   * Recibe 2 string como parámetros: el string que queremos buscar y el string que usaremos de reemplazo. * Retorna un nuevo string con el reemplazo. | Let frase = “Aguante Phyton!”;  Frase.replace(“Phyton”, “JS”) // devuelve “Aguante JS!” |

**OBJETOS LITERALES:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Let *nombre de la variable* = **{**  *nombrePropiedad* **:** *valor*  **}** | Un objeto es una estructura que puede contener propiedades y métodos. | Let auto = {  Patente : “AC 134 DD”  } |
|  | Si hay más de una propiedad, las separamos por comas. | Let tenista = {  nombre: “Roger”,  apellido: “Federer”  } |
|  | Si una propiedad almacena una función, diremos que es un **método** del objeto. | Let tenista = {  nombre: “Roger”,  edad: 38,  activo: true,  saludar : function (){  return “Hola, me llamo Roger”;  }  } |
| **Objeto.método()** | Para ejecutar un método de un objeto usamos la notación **objeto.método()** | Let tenista = {  nombre: “Roger”,  apellido: “Federer”,  saludar : function (){  return “Hola, me llamo Roger”;  }  }  Console.log (tenista.saludar()) |
| **This.propiedad** | La palabra reservada this hace referencia al objeto en sí donde estamos parados. Con la notación **this.propiedad** accedemos al valor de cada propiedad interna de ese objeto. | Let tenista = {  nombre: “Roger”,  apellido: “Federer”,  saludar : function (){  return “Hola me llamo “ + this.nombre;  }  } |

**FUNCIONES CONSTRUCTORAS:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Function *nombre, usamos mayúsculas en este caso* (*parámetros*) {  *Propiedades, con la notación this.propiedad definimos la propiedad del objeto que estamos creando en ese momento*  } | La función constructora nos permite armar un molde y luego crear todos los objetos que necesitemos. La función recibe un parámetro por cada propiedad que queramos asignarles al objeto. | Function Auto (marca, modelo) {  This.marca = marca;  This. Modelo = modelo;  } |
| Instanciar un objeto. **New**. | La función constructora **Auto()** espera los parámetros: marca y modelo. Para crear un objeto Auto debemos usar la palabra reservada **new** y llamar a la función pasándole los parámetros que espera. | Let miAuto = new Auto (“Ford”, “Falcon”); |

**ARROW FUNCTIONS:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Let *nombre de la variable que va a contener la función* = *parámetros, si hay uno solo no hace falta que vaya entre ()* **=>** *lo que retornamos* | Es una versión reducida de una function normal | Let sumar = (a, b) => a + b; |

**CICLOS FOR:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **Inicio:** Antes de arrancar el ciclo se establece el valor inicial de nuestro contador.  **Condición:** antes de ejecutar el código en cada vuelta, se pregunta si la condición resulta verdadera o falsa.  **Modificador:** incremento (++) o decremento (--) | Los ciclos nos permiten repetir instrucciones de manera sencilla. Podemos hacerlo una determinada cantidad de veces, o mientas se cumpla una condición. | For (let vuelta = 1; vuelta <= 5; vuelta++) {  Console.log (“Dando la vuelta número “+ vuelta)  } |

**CALLBACKS:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
|  | Un callback es una function que se pasa como parámetro de otra function. | Function nombreCompleto (nombre, apellido) {  Return nombre + “ “ + apellido  }  Function saludar (nombre, apellido, callback){  Return “!Hola “ + callback (nombre apellido) + “ !”  }  Saludar (“Juanito”, “Sanchez”, nombreCompleto) |

**MÉTODOS DEL ARRAY PARTE II:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **.map()**  array**.map(**function(elemento)**)**{  //definimos las modificaciones que queremos aplicar sobre cada elemento del array  } | Recibe una función como parámetro. Recorre el array y devuelve un nuevo array modificado. | Let números = [2, 4, 6];  Let elDoble = números.map (function (num){  Return num \*2  })  Console.log (elDoble); /[4, 8, 12] |
| **.filter ()**  Array**.filter(**function (elemento){  //definimos la condición que queremos utilizar.  //como filtro para cada elemento del array  }**)** | Recibe una función como parámetro. Recorre el array y filtra los elementos según una condición que exista en el callback.  Devuelve un nuevo array que contiene únicamente los elementos que hayan cumplido con esa condición. Es decir, que nuestro nuevo array puede contener menos elementos que el original. | Let edades = [22, 8, 17, 14, 30];  Let mayores = edades.filter (function (edad){  Return edad > 18  })  Console.log (mayores); // [22, 30] |
| **.reduce ()**  Array**.reduce (**function (acumulador, elemento) {  //definimos el comportamiento que queremos implementar sobre el acumulador y el elemento.  }**)** | Este método recorre el array y devuelve un único valor. Recibe un callback que se va a ejecutar sobre cada elemento del array. Este, a su vez, recibe dos parámetros: un acumulador y el elemento actual que esté recorriendo. | Let nums =[5, 7, 16];  Let suma = nums.reduce (function (acum, num) {  Return acum + num  })  Console.log (suma); //28 |
| **.forEach()**  Array**.forEach (**function (elemento){  //definimos el comportamiento que queremos  //implementar sobre cada elemento  }**)** | La finalidad de este método es iterar sobre un array.  Recibe un callback como parámetro y, a diferencia de los métodos anteriores, este **no retorna nada**. | Let países = [“Argentina”, “Cuba”, “Peru”];  Países.forEach (function (país) {  Console.log (país)  }) |
| **.concat()** | La finalidad de este método es unir dos arrays en uno. | Const array1 = [“a”, “b”, “c”];  Const array2 = [“d”, “e”];  Const array3 = array1.concant(array2); |

**OBJETO DATE Y SUS MÉTODOS:**

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Let miFecha = new Date ();  miFecha = es una variable para almacenar el objeto Date.  New = palabra reservada.  Método constructor = devuelve un objeto Date de JS. | Objeto para generar fechas |  |
| **.getDate()** | Retorna el número del día del mes de una fecha. Devolverá un número entre 1 y 31. | Let diaDeMiFecha = miFecha**.getDate ()**;  Console.log (diaDeMiFecha); //El nº del día de mi Fecha. |
| **.getMonth()** | Retorna el número del mes de una fecha. Devolverá un número entre 0 (enero) y 11 (diciembre) | Let mesDeMiFecha = miFecha**.getMonth ()**;  Console.log (mesDeMiFecha); //el nº del mes de mi fecha. |
| **.getDay** | Retorna el día de la semana de una fecha. Devolverá un número entre 0 (domingo) y 6 (sábado). | Let diaSemanaDeMiFecha = miFecha**.getDay ()**;  // el nº de día de la semana de miFecha |
| **.getFullYear ()** | Retorna el año completo de una fecha. Devolverá un número entre 1000 y 9999 | Let anioDeMiFecha = miFecha**.getFullYear ()**;  // el nº del año actual |
| **Fechas dinámicas** | Al instanciar a nuestro objeto, podemos pasarle 3 parámetros que representan, en orden: el **año completo**, el **mes** y el **día**. | Let miFechaCumple = new Date (1997, 9, 21) |

**DESTRUCTURING:** nos permite extraer datos de arrays y objetos literales.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| **Arrays:** | Para desestructurar un array declaramos una variable y, entre corchetes, escribimos el nombre que queremos. Luego, igualamos esa estructura al array del cual queremos extraer los datos. JS respeta el orden de los elementos del array, si queremos saltar un elemento dejamos un espacio vacío entre las “,”. | Let colores = [“Rojo”, “Azul”, “Amarillo”];  Let [rojo, azul, amarillo] = colores  Let [color1, , color3] = colores |
| **Objetos:** | Para desestructurar un objeto declaramos una variable y, entre llaves, declaramos el o los nombres de las propiedades que queremos extraer. A esa estructura la igualamos al objeto del cual queremos extraer los datos.  Si queremos cambiar el nombre de la variable tenemos que hacer a continuación de la propiedad “:” seguidos del nuevo nombre | Let persona = {nombre: “Laura”, edad: 31, faltas: 3};  Let {nombre, edad} = persona  Let {nombre, faltas**: totalFaltas**} = persona |

**SPREAD OPERATOR:** Este operador permite expandir cada uno de los datos de un elemento iterable dentro de otro elemento.

|  |  |  |
| --- | --- | --- |
| **Sintaxis** | **Uso** | **Ejemplo** |
| Se escribe con **3 puntos seguidos**, antecediendo al nombre de la variable.  …variable  Esta variable puede contener un array, un string, un objeto. |  |  |
| **Spread en arrays** | Copiamos todos los datos de un array en un array nuevo.  Como también podemos agregar todos los datos de un array, dentro de un array existente. | Let clubesUno = [“Boca”, “River”, “Racing”];  Let clubesDos = [“San lorenzo”, “Lanús”, “Gimnasia”];  Let todosLosClubes = […clubesUno, …clubesDos];  Let parte = [“los”, “cumplas”];  Let oración = [“Que”, …parte, “feliz”]; |
| **Spread en objetos** | Podemos copiar todas las propiedades de un objeto dentro de otro objeto existente. | Let auto = {marca: “Ferrari”, kms: 0, anio: 2019};  Let corredorUno = {nombre: “Vettel”, edad: 32, …auto};  Let corredorDos = {nombre: “Leclerc”, edad: 21, …auto};  Tanto corredorUno como corredorDos ahora tienen todas las propiedades que definimos en el objeto auto sin tener que definirlas a mano a cada uno de ellos. |
| **Spread y funciones** | Implementando este operador, podemos pasarle a una función un array como argumento. El operador **…** se encargará de expandir los datos para que la función los tome como argumentos separados.  Para ejemplificar usaremos el método de JS **Math.min()**, que recibe **N** cantidad de argumentos y devuelve el menor. | Let notas = [9.3, 8.5, 3.2, 7, 10];  Math.min(…notas); //devuelve 3.2 |