

Programación Imperativa

SEGUNDO BIMESTRE

PROFESOR: Martín Cejas
ncejas@digitalhouse.com
[Cronograma](#)

Clase 1: Módulo I - Pensamiento Computacional

¿Por qué Programación Imperativa?

Programar es crear soluciones creativas a un problema, es resolver una problemática, es descubrir vías de resolución de conflictos. En esta materia vamos a entender cómo funcionan los problemas lógicos y cómo descomponerlos para poder resolverlos de manera eficiente.

Pensamiento Computacional

Consiste en adaptar nuestra forma de razonar para poder resolver un problema lógico. Es importante entender cómo se tiene que pensar al programar para poder resolver un problema considerando que nuestro receptor no tiene “sentido común”, sino que solo recibe instrucciones precisas para ejecutarlas.

Es un proceso que permite identificar el problema, comprenderlo y plantear las soluciones más adecuadas.

También permite aplicar habilidades de comunicación, aprender a trabajar con otros para lograr un objetivo común, lidiar con problemas complejos y situaciones diversas, poner en juego la creatividad, desarrollar el pensamiento crítico y el razonamiento lógico.

“El pensamiento computacional implica resolver problemas, diseñando sistemas, y entender el comportamiento humano a partir de los conceptos fundamentales a la informática.

Pensamiento computacional incluye una gama de herramientas mentales que reflejan la amplitud del campo de la informática”. -Jeanette Wing

Etapas:

1. **Descomposición:** Se encarga de desarticular el problema en unidades más pequeñas, para que sean más fáciles de resolver.
2. **Reconocimiento de patrones:** Consiste en buscar similitudes o series que se repiten dentro o fuera de la situación problemática.
3. **Abstracción:** Simplificar un problema complejo, dejando de lado los datos poco relevantes para centrar la atención en los datos relevantes, y así poder definir un plan acotado.
4. **Algoritmo:** Se trata de la definición de los pasos ordenados necesarios para la solución del problema.

¿Cómo se puede aplicar el pensamiento computacional a un problema concreto?

1. Analizar el problema o desafío a resolver.
2. Tomar una decisión.
3. La solución: Nunca habrá una única solución.

**Si cambian las variables, debemos generar un nuevo algoritmo.*

Pensamiento computacional:

Proceso de resolución de problemas que se vale de la organización lógica y el análisis de datos. Posibilita el uso de una computadora u otro recurso para resolverlos.

Programar:

Proceso de diseñar y escribir una secuencia de instrucciones en un lenguaje que pueda ser entendido por una computadora.

Diagramas de flujo

El diagrama de flujo, flujograma o diagrama de actividades es la representación gráfica de un algoritmo o proceso. Se utiliza en disciplinas como programación, economía, procesos industriales y psicología cognitiva.

Se trata de una herramienta que nos permite determinar los pasos para poder llevar adelante un proceso o una tarea. Así, el diagrama de flujo resulta útil para pensar la resolución de un problema, sin importar qué lenguaje vayamos a usar.

Elementos

Línea de flujo (flecha)

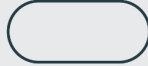
Forma ANSI/ISO



Muestra el orden de operación de los procesos. Se trata de una línea saliendo de un símbolo y apuntando a otro. Las flechas se agregan si el flujo no es el estándar de arriba hacia abajo, de izquierda a derecha.

Terminal

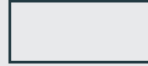
Forma ANSI/ISO



Indica el inicio o el fin de un programa o subproceso. Se representa como un óvalo. Usualmente, contiene la palabra "Inicio" o "Fin". También puede incluir alguna otra frase señalando el comienzo o cierre de un proceso, como "presentar consulta" o "recibir producto".

Proceso

Forma ANSI/ISO



Representa un conjunto de operaciones que cambian el valor, la forma o la ubicación de datos. Representado como un rectángulo.

Decisión

Forma ANSI/ISO



Muestra una operación condicional que determina cuál de los dos caminos tomará el programa. La operación es, en general, una pregunta que tiene como respuesta "sí/no" o "verdadero/falso". Representada como un rombo.

Entrada

Forma ANSI/ISO



Representado como un paralelogramo, indica los datos que se ingresan o extraen del proceso.

Salida

Forma ANSI/ISO



Indica el proceso de hacer salir datos, en la forma de mostrar resultados. Representado como una hoja de papel impresa.

Anotación (comentario)

Forma ANSI/ISO



Indica información adicional acerca de un paso en el programa. Representado como un rectángulo abierto con una línea —que puede ser punteada— conectándolo con el símbolo correspondiente del diagrama de flujo.

Conector de página

Forma ANSI/ISO



Pares de conectores etiquetados reemplazan líneas largas o confusas en la página del diagrama. Representados como pequeños círculos con una letra dentro.

Clase 2: Módulo I - Pensando como la computadora

Hacer un Origami

Un algoritmo es una serie de pasos para resolver un problema.

¿Cuál es la diferencia con una computadora? Que, por ejemplo, si nos estamos lavando los dientes y no hay agua o nos quedamos sin pasta de dientes, las personas pueden buscar soluciones basadas en experiencias anteriores o improvisar una solución, en el caso de las computadoras, si un paso programado no se puede cumplir, ese proceso no avanzará y quedará inconcluso.

Paradigmas de programación - del símbolo al texto

¿Qué son los paradigmas de programación?

Diferentes estilos de usar la programación para resolver un problema.

Programación Imperativa

Los programas consisten en una sucesión de instrucciones o conjunto de sentencias, paso a paso. Dentro del paradigma imperativo encontramos estos enfoques:

- **Programación estructurada:** La programación estructurada es un tipo de programación imperativa donde el flujo de control se define mediante bucles anidados, condicionales y subrutinas, en lugar de a través de GO TO.
- **Programación procedimental:** Este paradigma de programación consiste en basarse en un número muy bajo de expresiones repetidas, englobadas todas en un procedimiento o función y llamarlo cada vez que tenga que ejecutarse.
- **Programación Modular:** Consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más manejable y legible. Se trata de una evolución de la programación estructurada para resolver problemas de programación más complejos.

Programación Declarativa

Este paradigma no necesita definir algoritmos puesto que describe el problema en lugar de encontrar una solución al mismo. Así, utiliza el principio del razonamiento lógico para responder a las preguntas o cuestiones consultadas.

Se divide en dos:

- **Programación lógica:** Un programa puede ser descrito definiendo ciertas relaciones entre conjuntos de objetos, a partir de las cuales otras pueden ser calculadas empleando deducción.
- **Programación funcional:** Se basa en la evaluación de expresiones, no en la ejecución de instrucciones.

Programación Orientada a Objetos

Se construyen modelos de objetos que representan elementos (objetos) del problema a resolver, que tienen características y funciones. Permite separar los diferentes componentes de un programa. Se acerca de alguna manera a cómo expresaríamos las cosas en la vida real.

Programación Reactiva

Se basa en escuchar lo que emite un evento o cambios en el flujo de datos, en donde los objetos reaccionan a los valores que reciben de dicho cambio.

Los sistemas reactivos deben ser:

- **Responsivos:** aseguran la calidad del servicio cumpliendo unos tiempos de respuesta establecidos.
- **Resilientes:** se mantienen responsivos incluso cuando se enfrentan a situaciones de error.
- **Elásticos:** se mantienen responsivos incluso ante aumentos en la carga de trabajos.
- **Orientados a mensajes:** minimizan el acoplamiento entre componentes al establecer interacciones basadas en el intercambio de mensajes de manera asíncrona.

Comunicación: Transmisión de señales de un emisor y a un receptor mediante un código común (el lenguaje).

Órdenes mediante símbolos (lightbot)

Un símbolo carga un significado. Una computadora es una máquina que los interpreta, gracias a los significados puestos por el humano.

Sintaxis: modo estructurado de combinar los símbolos y de asignarles significados.

Lenguaje y ambigüedad

El cerebro humano puede interpretar estas ambigüedades, resignificarlas hasta que tengan sentido dentro del contexto.

Las computadoras —por el momento— no pueden hacer eso. Son máquinas muy complejas que necesitan reglas simples. No debe haber ambigüedad.

Node.js

Es un entorno de ejecución que nos permite ejecutar JavaScript por fuera de un navegador web.

Arquitectura de Node.js

Todos los navegadores presentan un motor de JavaScript para leer y renderizar (representar gráficamente) código de JavaScript. Esto hace que el lenguaje dependa sí o sí de un navegador para poder ejecutarse.

Los navegadores utilizan distintos motores y es por esta variedad que a veces un mismo código de JavaScript puede comportarse de manera diferente dependiendo del navegador en el que se esté ejecutando.



Node.js está construido bajo el motor v8 de Google Chrome. Esto lo convierte en un entorno de ejecución para JavaScript y logra que el lenguaje deje de depender del navegador para poder ejecutarse.

De esta forma, podemos programar tanto el front-end como el back-end en un mismo lenguaje: JavaScript.



Clase 3: Módulo I - Cierre de semana

Clase 4: Módulo II - Variables, tipos de datos y operadores

[Prompt sync](#)

Variables

Podemos pensar las variables de dos maneras distintas. Hay quienes prefieren verlas como nombres de algo, o formas de nombrar algo, y hay quienes eligen pensarlas como cajas vacías, con etiquetas para diferenciarlas —sus nombres—, cuyo interior es algo así como cualquier cosa que queramos poner.

Es un espacio en memoria donde se almacena un dato que podemos utilizar a futuro.

Para poder utilizar las variables en JS, debemos usar **Palabras reservadas**:

- No escribir tildes
 - No dejar espacios
 - No empezar con mayúscula
- **let** + nombre de la variable = ... No puede ser redefinida en el mismo bloque de código.

Declaración de una variable

```
let nombreSignificativo;
```



let

La palabra reservada **let** le indica a JavaScript que vamos a **declarar una variable de tipo let**.

Nombre

- Solo puede estar formado por letras, números y los símbolos \$ (pesos) y _ (guion bajo).
- No pueden empezar con un número.
- No deberían contener ñ o caracteres con acentos.



Es una buena práctica que los nombres de las variables usen el formato Camel case, como `variableEjemplo` en vez de `variableejemplo` o `variable_ejemplo`.

La primera vez que declaramos una variable es necesaria la palabra reservada `let`. Una vez que la variable ya fue declarada, le asignamos valores sin `let`.

- **const**: Una vez definido su valor, no puede ser modificado.

Declaración con const

Las variables **const** se declaran con la palabra reservada **const**.

```
{ } const email = "mi.email@hotmail.com";
```

Las variables declaradas con **const** funcionan igual que las variables **let**, estarán disponibles solo en el bloque de código en el que se hayan declarado.

Al contrario de **let**, una vez que les asignemos un valor, no podremos cambiarlo.

```
{ } email = "mi.otro.email@hotmail.com";  
// Error de asignación, no se puede cambiar  
// el valor de un const
```

- **var**: Ya no se utiliza

var

```
if (true) {  
  var nombre = "Juan";  
}  
  
console.log(nombre);  
// Ok, muestra "Juan"
```

Cuando usamos **var**, JavaScript ignora los bloques de código y convierte nuestra variable en global.

Eso quiere decir que si hay otra variable **nombre** en nuestro código, seguramente estemos pisando su valor.

let

```
if (true) {  
  let nombre = "Juan";  
}  
  
console.log(nombre);  
// Error: nombre no existe
```

Cuando usamos **let**, JavaScript respeta los bloques de código. Eso quiere decir que nombre no podrá ser accedida fuera del **if**.

También quiere decir que podemos tener variables con el mismo nombre en diferentes bloques de nuestro código.

Preguntar en clase: Es contradictorio con lo visto en el módulo.

¡Genial, lo lograste! Recordar que es una buena práctica nombrar las constantes con MAYÚSCULAS.

Tipos de Datos

Los tipos de datos le permiten a JavaScript conocer las características y funcionalidades que estarán disponibles para ese dato.

- Numéricos (number)
- Cadenas de caracteres (string)
- Lógicos o booleanos (boolean)
- Undefined (valor sin definir): Indica la ausencia de valor. Las variables tienen un valor indefinido hasta que les asignamos uno.
- Null (valor nulo): Lo asignamos nosotros para indicar un valor vacío o desconocido.
- Array y Objeto literal
- La propiedad global NaN es un valor de tipo numérica que representa Not-A-Number.

Operadores

Los operadores nos permiten manipular el valor de las variables, realizar operaciones y comparar sus valores.

1. **De asignación:** Asignan el valor de la derecha en la variable de la izquierda.

```
{ } let edad = 35; // Asigna el número 35 a edad
```

2. **Aritméticos:** Nos permiten hacer operaciones matemáticas, devuelven el resultado de la operación.

```
{ } 10 + 15 // Suma → 25
      10 - 15 // Resta → -5
      10 * 15 // Multiplicación → 150
      15 / 10 // División → 1.5
```

```
{ } 15++ // Incremento, es igual a 15 + 1 → 16
      15-- // Decremento, es igual a 15 - 1 → 14
```

```
{ } 15 % 5 // Módulo, el resto de dividir 15 entre 5 → 0
      15 % 2 // Módulo, el resto de dividir 15 entre 2 → 1
```

**El operador de módulo % nos devuelve el resto de una división.*

3. **De concatenación:** Sirven para unir cadenas de texto. Devuelven otra cadena de texto.

```
{ } let nombre = 'Teodoro';
      let apellido = 'García';
      let nombreCompleto = 'Me llamo ' + nombre + ' ' + apellido;
```

Template literals

Existe otra forma de armar strings a partir de variables, y es con los template literals utilizando backtick en lugar de comillas y las variables entre llaves a continuación del símbolo \$. En este ejemplo lo escribiríamos así: `Me llamo ${nombre} ${apellido}`

4. **De comparación simple:** Comparan dos valores, devuelven verdadero o falso.

```
{ } 10 == 15 // Igualdad → false
      10 != 15 // Desigualdad → true
```

5. **De comparación estricta:** Comparan el valor y el tipo de dato también.

```
{ } 10 === "10" // Igualdad estricta → false
      10 !== 15 // Desigualdad estricta → true
```

6. De comparación: Comparan dos valores, devuelven verdadero o falso.

```
{  
  15 > 15 // Mayor que → false  
  15 >= 15 // Mayor o igual que → true  
  10 < 15 // Menor que → true  
  10 <= 15 // Menor o igual que → true  
}
```

**Siempre debemos escribir el símbolo mayor (>) o menor (<) antes que el igual (>= o <=). Si lo hacemos al revés (=> o =<), JavaScript lee primero el operador de asignación = y luego no sabe qué hacer con el mayor (>) o el menor (<).*

[JSBIN](#)

Clase 5: Módulo II - Trabajando con Funciones

Función: Listado de procedimientos que se va a ejecutar cuando sea necesario. Su objetivo es tener toda la lógica necesaria para cumplir con un objetivo definido. Una función es un bloque de código que nos permite agrupar una funcionalidad para usarla todas las veces que necesitemos.

Normalmente, realiza una tarea específica y retorna un valor como resultado.

Pueden ser:

- **Expresadas:** Es aquella que se asigna como valor a una variable. Se carga antes de que cualquier código sea ejecutado.

```
let sumar = function () {...}
```

- **Declaradas:** Es aquella que recibe un nombre formal y no se asigna como valor a una variable. Se carga únicamente cuando el intérprete alcanza la línea de código donde se encuentra la función.

```
function sumar () {...}
```

**Palabras reservada: function, return*

```

JS funciones.js > ...
1  // Función expresada
2
3  let sumar = function(numeroA, numeroB){
4      return numeroA + numeroB;
5  }
6
7  console.log(sumar(7, 9));
8
9  // Función declarada
10
11 function restar(numeroC, numeroD){
12     return numeroC - numeroD;
13 }
14
15 console.log(restar(10, 3));

```

Scope: Alcance que llega a tener una variable. Existen dos tipos:

- **Local:** Cuando existen variables declaradas exclusivamente dentro de una función. Fuera de la función la variable es inexistente.

```

function hola () {
    // variable local
    let saludo = "Hola ¿qué tal?";
    return saludo;
}

console.log(hola()); // "Hola ¿qué tal?"
console.log(saludo); // Undefined Variable

```

- **Global:** Cuando las variables se declaran fuera de cualquier función. Teniendo así un alcance a ellas en cualquier lugar del código, incluso dentro de funciones.

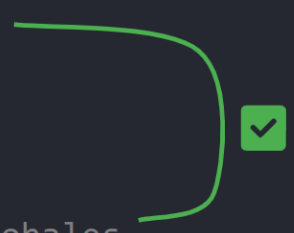
```

let elMejorLenguaje = "Javascript, es lo más";
function estoyAprendiendo () {
    return elMejor Lenguaje;
}

console.log(estoyAprendiendo());
// "Javascript, es lo más"

```

```
{  
  // todo el código que escribamos fuera  
  // de las funciones es global  
  function miFuncion() {  
    // Desde adentro de las funciones  
    // Tenemos acceso a las variables globales  
  }  
}
```



**Una variable con un scope local tiene predominancia sobre una con un scope global.*

Nombre de la función: Definimos un nombre para referirnos a nuestra función al momento de querer invocarla.

Durante la declaración nos ocupamos de construir la máquina y durante la invocación la ponemos a funcionar.

Antes de poder invocar una función, necesitamos que haya sido declarada.

La forma de invocar —también se puede decir llamar o ejecutar— una función es escribiendo su nombre seguido de apertura y cierre de paréntesis.

Parámetros: Escribimos los paréntesis y, dentro de ellos, los parámetros de la función. Si hay más de uno, los separamos usando comas ,.

Si la función no lleva parámetros, igual debemos escribir los paréntesis sin nada adentro ().

Si la función tiene parámetros, se los podemos pasar dentro de los paréntesis cuando la invocamos. Es importante respetar el orden porque JavaScript asignará los valores en el orden en que lleguen.

También es importante tener en cuenta que, cuando tenemos parámetros en nuestra función, JavaScript va a esperar que se los indiquemos al ejecutarla.

- **Parámetros (cont.):** Dentro de nuestra función podremos acceder a los parámetros como si fueran variables. Es decir, con solo escribir los nombres de los parámetros,

podremos trabajar con ellos.

```
{  
  function sumar (a, b) {  
    return a + b;  
  }  
}
```

- Podemos definir valores por defecto. Si agregamos un igual = luego de un parámetro, podremos especificar su valor en caso de que no llegue ninguno.

```
{  
  function saludar(nombre = 'visitante',  
    apellido = 'anónimo') {  
    return 'Hola ' + nombre + ' ' + apellido;  
  }  
  
  saludar(); // retornará 'Hola visitante anónimo'  
}
```

Cuerpo: Entre las llaves de apertura y de cierre escribimos la lógica de nuestra función, es decir, el código que queremos que se ejecute cada vez que la invoquemos.

El retorno: Es muy común, a la hora de escribir una función, que queramos devolver al exterior el resultado del proceso que estamos ejecutando.

*Para eso utilizamos la palabra reservada **return** seguida de lo que queramos retornar.*

Guardando los resultados

En caso de querer guardar lo que retorna una función, será necesario almacenarlo en una variable.

```
function hacerHelados(cantidad) {
    return '🍦'.repeat(cantidad);
}

let misHelados = hacerHelados(3);
console.log(misHelados); // Mostrará en consola '🍦🍦🍦'
```

Llamamos **parámetros** a las variables que escribimos cuando definimos la función. Llamamos **argumentos** a los valores que enviamos cuando invocamos la función.

Arrow Functions

Las arrow functions —o funciones arrow— son una forma de crear funciones incorporadas a partir de ES6 —ECMAScript versión 6—. Una de sus ventajas es que son más concisas que las funciones clásicas creadas con la palabra reservada function. Funciones de sintaxis compacta.

```
// forma clásica
function sumar(a, b) {
    return a + b;
}
console.log( sumar(2, 4) ); // 6

// ES6 arrow function
let sumar = (a, b) => a + b;

console.log( sumar(2, 4) ); // 6
```

Es necesario que sean asignadas como texto simple para destacar.

*Su operador se llama operador flecha.

No son necesarias las llaves ni las palabras reservadas: return, function.

Si tiene varias sentencias de código a ejecutar si será necesarias las llaves y la palabra reservada return.

```

JS arrow-functions.js > [❌] dividir
1  let laMitad = numero => numero / 2;
2
3  console.log(laMitad(8));
4
5  let dividir = (numeroA, numeroB) => numeroA / numeroB;
6
7  console.log(dividir(20, 4));
8

```

```

let horaActual = () => {
  let fecha = new Date();
  return fecha.getHours() + ':' +
    fecha.getMinutes();
}

```

	var	const	let
scope	global or local	block	block
redeclare?	yes	no	no
reassign?	yes	no	yes
hoisted?	yes	no	no

© 2019

Funcion Flecha	<pre>const sumar = (a,b) => a+b;</pre>
Funcion Expresada	<pre>const sumar = function(a, b){ return a + b; }</pre>
Funcion Declarada	<pre>function sumar(a,b){ return a + b }</pre>

Clase 6: Módulo II - Cierre de semana

[Link](#) *parseInt* y *Number* para transformar strings en numbers

[Link](#) Instalación y uso de *prompt*

Clase 7: Módulo II - Controlando el flujo de la aplicación

Operadores Lógicos

Un operador es aquella porción de código que nos permitirá hacer operaciones aritméticas, comparaciones, concatenaciones, entre otras cosas más.

- **Operador de asignación (=):** Nos permite asignar un valor a una variable determinada.

```
let paisDeNacimiento = 'Argentina';
```

- **Operadores aritméticos:** Nos permiten hacer las operaciones matemáticas tradicionales.
 - Suma (+)
 - Resta (-)
 - Multiplicación (*)
 - División (/)
 - Módulo (%): Retorna el resto o residuo de una división.
 - Incremento (++): Sumar 1 unidad a un número determinado.
 - Decremento (--): Restar 1 unidad a un número determinado.
- **Operadores de Comparación:**
 - **Simple (== / !=):** Si un valor es igual a otro. Como respuesta nos trae un dato booleano (true o false).
 - **Estricta (=== / !==):** Si un valor y tipo de dato es igual a otro. También trae un booleano como respuesta.
 - **Mayor que (>) o igual (>=)**
 - **Menor que (<) o igual (<=)**
- **Operadores lógicos:** Nos permiten unir sentencias de código por las cuales queremos preguntar.
 - **AND (&&):** Para preguntar por varias sentencias a la vez

```
let dia = "domingo"  
let clima = "soleado"
```



```
dia == "domingo" && clima == "soleado"  
// true
```

- **OR(| |)**: Se implementa usando dos pipe. Sirve para preguntar por varias sentencias a la vez, pero a diferencia del AND, si una sentencia se evalúa como verdadera, toda la sentencia será verdadera.



`8 >= 15` ✗ `8 != 3` ✓

`8 >= 15 || 8 != 13 // true`

- **Negación NOT (!)**: Nos permite cambiar el valor de falso o verdadero, dependiendo de cómo era el valor anteriormente.
- **Operador de Concatenación**: Sirve para unir 2 o más cadenas de texto en una sola.

Cuestionario teórico: Operadores lógicos

1. **¿Qué tienen en común los operadores lógicos con los de comparación?** Ambos devuelven un valor booleano como resultado.
2. **¿En qué se diferencia la comparación simple de la estricta?** La comparación estricta compara el valor y el tipo de dato, mientras que la débil solo el valor.
3. **¿Cuál es la función del símbolo "!"?** Es el símbolo de negación, niega cualquier condición o sentencia que hagamos.
4. **¿Por qué debemos escribir el símbolo > o < antes de =?** Porque de lo contrario JavaScript lee primero el = como asignación y luego no sabe cómo continuar.
5. **¿Cuál es la principal diferencia entre el “y lógico” (&&) y el “o lógico” (| |) para obtener un resultado “true”?** En el && ambas condiciones deben ser correctas, mientras que en el | | una condición puede ser falsa.

Cuestionario: ¿Qué retornan las siguientes operaciones lógicas?

1. **false | | true // true** El operador | | analiza de izquierda a derecha y retorna cuando encuentra un valor verdadero o el último valor de la sentencia.
2. **false | 3 == 4 // 0** Si bien JavaScript lee e interpreta la sentencia y retorna un 0, hay un error de sintaxis, ya que la forma correcta de usar los operadores lógicos es con símbolos dobles ("| |" y "&&").
3. **false && (3 == 4) // false** El operador && devuelve el primer valor o expresión analizado como false.

-
4. `10 >= 15 && 10 !== 11` // false Como ven, no es necesario los paréntesis en esta situación, ya que el operador de comparación es de mayor prioridad que el operador lógico &&.
 5. `12 % 2 == 0 && 12 !== 21` // true El operador de módulo va a devolver siempre el resto de la división, y el operador de negación niega cualquier comparación.
 6. `(8-15 == 8 || 7>6 = -2)` // Error Recordá que las comparaciones son con `==` o `===`. Un solo `=` es simplemente una asignación de un valor a una variable. Por lo tanto, se produce un error al querer asignarle un valor a otro valor, resultando en el error: "SyntaxError: Invalid left-hand side in assignment" que refiere a que no se puede utilizar valores en el lado izquierdo de las asignaciones.
 7. `3+5 == "8" && 5-4 === 1` // true
 8. `'Zapato' == 'trampa' || "hola" <= "chau"` // false Cuando se comparan strings, el `==` compara valor y `<=` compara por orden alfabético. Ejemplo: `console.log("a" < "bbbbbb");` //true
 9. `"Gato" && "Perro"` // Perro En el caso de los strings y el operador &&, al ser ambos strings verdaderos (ya que tienen una cadena de caracteres), la respuesta es el último string de la sentencia. Opciones (para single choice y multiple choice)
 10. `"Gato" || "Perro"` // Gato A diferencia del operador &&, en este caso —al tener que cumplirse solo una condición—, como ambos son true, la respuesta es la primera sentencia true encontrada.

Condicionales

Nos permiten evaluar condiciones y realizar diferentes acciones según el resultado de dicha evaluación.

Condición if: Permite ejecutar un bloque de código siempre que se cumpla con una condición.

anatomía del if

if (condición o condiciones) {código a ejecutar cada vez que la condición se cumpla}

Else: Código a ejecutar si la condición dentro del *if*, no se cumple.

Else if: Permite crear un escenario para cuando no se cumplen las condiciones del *if* inicial, pero sin tener que ejecutar el bloque de código del *else*.

**Necesita de una o varias sentencias a evaluar.*

```
1  if (clima == "soleado" && dia == "domingo") {  
2      console.log("Daré un paseo en la plaza");  
3  } else if (clima == "nublado" && dia == "sábado") {  
4      console.log("Iré al cine");  
5  } else {  
6      console.log("Quizá mejor me quedo en casa");  
7  }
```

Tené presente

Que un *if* no siempre necesita de un *else* o *else if*.

Que puede tener muchos *else if* si así lo quisieras.

Que siempre que implementes un *else*, deberá existir solo uno de ellos.

Componentes de un *if*

Nos permiten evaluar condiciones y realizar diferentes acciones según el resultado de esas evaluaciones.

-
1. **Condicional simple:** Versión más básica del *if*. Establece una condición y un bloque de código a ejecutar en caso de que sea verdadera.

```
{ } if (condición) {  
    // código a ejecutar si la condición es verdadera  
}
```

2. **Condicional con bloque else:** Igual al ejemplo anterior, pero agrega un bloque de código a ejecutar en caso de que la condición sea falsa.

*Es importante tener en cuenta que el bloque *else* es opcional.

```
{ } if (condición) {  
    // código a ejecutar si la condición es verdadera  
} else {  
    // código a ejecutar si la condición es falsa  
}
```

3. **Condicional con bloque else if:** Igual que el ejemplo anterior, pero agrega un *if* adicional. Es decir, otra condición que puede evaluarse en caso de que la primera sea falsa.

Podemos agregar todos los bloques *else if* que queramos, solo uno podrá ser verdadero. De lo contrario, entrará en acción el bloque *else*, si existe.

```
{ } if (condición) {  
    // código a ejecutar si la condición es verdadera  
} else if (otra condición) {  
    // código a ejecutar si la otra condición es verdadera  
} else {  
    // código a ejecutar si todas las condiciones son falsas  
}
```

Funcionamiento de un if

```
let edad = 19;
let acceso = '';

if (edad < 16) {
  acceso = 'prohibido';
} else if (edad >= 16 && edad <= 18) {
  acceso = 'permitido solo acompañado de un mayor';
} else {
  acceso = 'permitido';
}
```

if ternario y switch

Son condicionales que nos permiten consultar si un bloque de código cumple con cierta condición o no, para posteriormente ejecutar otro bloque de código o no.


if ternario

- No lleva llaves que encierran los bloques de código {}
- No lleva la palabras reservadas *if* ni *else*
- Se escribe 100% de forma horizontal, en una sola línea.
- Sólo incluye el escenario de cuando la condición se cumple y cuando no se cumple.

```
condición ? expresión para el true : expresión para el false;
```

IMPORTANTE

- Para el *if ternario* es obligatorio poner código en la segunda expresión. Si no queremos que pase nada, podemos usar un string vacío "".
- Se suele asignar a una variable el resultado de ese *if ternario*.



```
1 let elMayor = 4 > 10 ? "El 4 es mayor" : "El 10 es mayor";
2 console.log(elMayor); // El 10 es mayor
```

Switch

Preguntar por algo → si ese algo es verdadero → ejecutar un bloque de código

```
switch (expresión) {  
  case caso1:  
    console.log("se cumple el caso 1");  
    break;  
}
```

```
1 let dia = "domingo";  
2 switch (dia) {  
3   case "lunes":  
4     console.log("Es lunes, se labura");  
5     break;  
6   case "sábado":  
7     console.log("Es sábado, se sale");  
8     break;  
9   case "domingo":  
0     console.log("Es domingo, se come asado");  
1     break;  
2   default:  
3     console.log("No es lunes ni sábado ni domingo");  
4     break;  
5 }
```

Palabras reservadas:

- **switch:** condicional
- **case:** caso de condicional
- **break:** si el caso se cumple, detener la ejecución.
- **default:** caso que se ejecuta si ninguno de los casos coincide con la expresión que estamos evaluando.

***PREGUNTAR EN CLASE:** Ejemplo 2 de If ternario / Switch, no funciona en VSC.

SI FUNCIONA! Reemplacé console.log por return, y el break me sale unreachable. En estos casos está ok que no se use?

Clase 8: Módulo II - Ciclos

Un ciclo —ya sea en programación o en la vida cotidiana— es una serie de estados por los que pasa un acontecimiento o fenómeno, que se repiten siempre en el mismo orden. En programación, para hacer que nuestro código se siga ejecutando, mientras una condición se cumpla, utilizamos el `for`.

Ciclos FOR

Los ciclos nos permiten repetir instrucciones de manera sencilla. Podemos hacerlo una determinada cantidad de veces o mientras se cumpla una condición.

Estructura básica

Consta de 3 partes que definimos dentro de los paréntesis. En conjunto, nos permiten determinar de qué manera se van a realizar las repeticiones y definir las instrucciones que queremos que se lleven a cabo en cada una de ellas.

```
{} for (inicio; condición ; modificador) {  
    //código que se ejecutará en cada repetición  
}
```

```
{} for (let vuelta = 1; vuelta <= 5; vuelta++) {  
    console.log('Dando la vuelta número ' + vuelta);  
};
```

- **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.
 - Si es verdadera, continúa con nuestras instrucciones.
 - Si es falsa, detiene el ciclo.

- **Modificador —incremento o decremento—:** Luego de ejecutar nuestras instrucciones, se modifica nuestro contador de la manera que hayamos especificado. En este caso se le suma 1, pero podemos hacer la cuenta que queramos.

El ciclo for en acción

En cada ciclo se verifica si el valor de vuelta es menor o igual a 5. Si es así, se ejecuta el `console.log()` y se incrementa el valor de vuelta en 1.

Cuando vuelta deje de ser menor o igual a 5, se corta el ciclo.

Iteración #	Valor de vuelta	¿Vuelta <= 5 ?	Ejecutamos
1	1	true	✓
2	2	true	✓
3	3	true	✓
4	4	true	✓
5	5	true	✓
6	6	false	✗

Preguntar en clase: Ejercicio 2: Estructura for 2, no está del todo claro.

Ciclos: while y do while

En este caso veremos otras dos formas de representar y recrear acciones cíclicas: el while loop y el do while.

Estructura básica while

Tiene una estructura similar a la de los condicionales if/else, palabra reservada + condición entre paréntesis. Sin embargo, el while Loop reevalúa esa condición repetidas veces y ejecuta su bloque de código hasta que la condición deja de ser verdadera.

```
while (condicion) {  
    //código que se ejecutará en cada repetición  
}  
    // Hace algo para que la condición eventualmente se deje  
    de cumplir  
}
```

```
let vuelta = 1  
while(vuelta <= 5) {  
    console.log('Dando la vuelta número ' + vuelta);  
    vuelta++ //al final de cada vuelta sumara 1 a vuelta  
};
```

Antes de ejecutar el código en cada vuelta, se pregunta si la condición resulta verdadera o falsa.

- Si es verdadera, continúa con nuestras instrucciones.
- Si es falsa, detiene el ciclo.

Es importante generar el **contador** al comenzar para evitar caer en lo que se conoce como *loop infinito*.

El **loop infinito** sucede cuando nuestra condición es constantemente verdadera, lo que resulta en ejecutar nuestro código eternamente. Esto puede causar varios problemas, siendo el más importante el que trabe todo nuestro programa.

El ciclo **do while** es similar al **while**, pero se diferencia en que, sin importar la condición, la acción se realizará al menos una vez.

Estructura básica do while

A diferencia del ciclo while, la condición en este caso se verifica al finalizar el bloque de código. Por lo tanto, sin importar lo que se resuelva, las acciones se realizarán al menos una vez.

Fuera de esto, el ciclo do while es idéntico en funcionalidad al ciclo while.

```
let vuelta = 5
do{
  console.log('Dando la vuelta número ' + vuelta);
  vuelta++ //Se suma 1 a vuelta por lo tanto vuelta = 6
} while(vuelta <= 5); //al vuelta ser 6 la condición retorna
false y se termina el bloque de código
```

Clase 9: Módulo II - Cierre de semana

Ejercicio de clase 7

¿Puede subir?

En un parque de diversiones nos piden un programa para verificar si los pasajeros de la montaña rusa pueden subir al juego.

Crear una función puedeSubir() que reciba dos parámetros: altura de la persona y si viene acompañada. Debe retornar un valor booleano (TRUE, FALSE) que indique si la persona puede subirse o no, basado en las siguientes condiciones:

- Debe medir más o igual de 1,40m y menos de 2 metros.
- Si mide menos de 1,40m hasta 1.20m, deberá venir acompañado.
- Si mide menos de 1,20m, no podrá subir ni acompañado.

Modificar la función para impedir la subida al juego si la persona fue penalizada por no respetar las normas y reglas del juego.

Clase 10: Módulo II - Strings y Arrays

¿Cómo podemos agrupar mucha información en una sola variable? Para ello contamos con un tipo de dato un poco más estructurado llamado **array** —también conocido como lista o arreglo—, que no es más que una “colección de elementos”.

Los [arrays](#) nos permiten generar una colección de datos ordenados.

Estructura de un array

Utilizamos corchetes [] para indicar el inicio y el fin de un array. Y usamos comas , para separar sus elementos.


Dentro de un array, podemos almacenar la cantidad de elementos que queramos, sin importar el tipo de dato de cada uno. Es decir, podemos tener en un mismo array datos de tipo string, number, boolean y todos los demás.

```
{ } let miArray = ['Star Wars', true, 23];
```

Posiciones dentro de un array

Cada dato de un array ocupa una posición numerada conocida como índice. La primera posición de un array es siempre 0.

```
{ } let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```



Para acceder a un elemento puntual de un array, nombramos al array y, dentro de los corchetes, escribimos el índice al cual queremos acceder.

```
{ } pelisFavoritas[2];  
// accedemos a la película Alien, el índice 2 del array
```

Longitud de un array

Otra propiedad útil de los arrays es su longitud, o cantidad de elementos. Podemos saber el número de elementos usando la propiedad length.

```
{ } let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```

1 + 1 + 1 = 3

Para acceder al total de elementos de un array, nombramos al array y, seguido de un punto (.), escribiremos la palabra `length`.

```
{ } pelisFavoritas.length;  
// Devuelve 3, el número de elementos del array
```

Métodos de arrays

Los arrays en JavaScript tienen muchas funciones llamadas métodos que nos van a permitir manipular los datos presentes en ese arreglo.

Pop para extraer

✕ ✕

Push para insertar elementos al final

✕ ✕

Shift para extraer

Unshift para insertar elementos al principio de un array

IndexOf / lastIndexOf para preguntar el índice de una ocurrencia

Join para unificar todos los elementos presentes en un array

Para JavaScript, los arrays son un tipo especial de objetos. Por esta razón disponemos de muchos métodos muy útiles a la hora de trabajar con la información que hay adentro.

Una función es un bloque de código que nos permite agrupar funcionalidad para usarla muchas veces. Cuando una función pertenece a un objeto, en este caso nuestro array, la llamamos [método](#).

- **.push()** : Agrega uno o varios elementos al final del array.
 - Recibe uno o más elementos como parámetros.
 - Retorna la nueva longitud del array.

```

let colores = ['Rojo', 'Naranja', 'Azul'];
colores.push('Violeta'); // retorna 4
console.log(colores); // ['Rojo', 'Naranja', 'Azul', 'Violeta']
{}
colores.push('Gris', 'Oro');
console.log(colores);
// ['Rojo', 'Naranja', 'Azul', 'Violeta', 'Gris', 'Oro']

```

- **.pop()** : Elimina el último elemento de un array.
 - No recibe parámetros.
 - Devuelve el elemento eliminado.

```

let series = ['Mad Men', 'Breaking Bad', 'The Sopranos'];

// creamos una variable para guardar lo que devuelve .pop()
{} let ultimaSerie = series.pop();

console.log(series); // ['Mad men', 'Breaking Bad']
console.log(ultimaSerie); // ['The Sopranos']

```

- **.shift()** : Elimina el primer elemento de un array.
 - No recibe parámetros.
 - Devuelve el elemento eliminado.

```

let nombres = ['Frida', 'Diego', 'Sofía'];

// creamos una variable para guardar lo que devuelve .shift()
{} let primerNombre = nombres.shift();

console.log(nombres); // ['Diego', 'Sofía']
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']

marcas.unshift('Ferrari', 'BMW');
console.log(marcas); // ['Ferrari', 'BMW', 'Ford', 'Audi']
```

- **.join()**: Une los elementos de un array utilizando el separador que especifiquemos. Si no 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 = dias.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 un -1.


```

let frutas = ['Manzana', 'Pera', 'Frutilla'];
frutas.indexOf('Frutilla');
// Encontró lo que buscaba. Devuelve 2, el índice del elemento

frutas.indexOf('Banana');
// No encontró lo que buscaba. Devuelve -1

```

- **.lastIndexOf()** : Similar a .indexOf(), con la salvedad de que empieza buscando el elemento por el final del array (de atrás hacia adelante).
En caso de haber elementos repetidos, devuelve la posición del primero que encuentre (o sea el último si miramos desde el principio).

```

let clubes = ['Racing', 'Boca', 'Lanús', 'Boca'];

clubes.lastIndexOf('Boca');
// Encontró lo que buscaba. Devuelve 3

clubes.lastIndexOf('River');
// No encontró lo que buscaba. Devuelve -1

```

- **.includes()** : También similar a .indexOf(), con la salvedad que retorna un booleano.
 - Recibe un elemento a buscar en el array.
 - Retorna true si encontró lo que buscábamos, false en caso contrario.

```

let frutas = ['Manzana', 'Pera', 'Frutilla'];

frutas.includes('Frutilla');
// Encontró lo que buscaba. Devuelve true

frutas.includes('Banana');
// No encontró lo que buscaba. Devuelve false

```

Strings

Para JavaScript un string no es más que una colección de caracteres.

Propiedades y métodos de strings

Para JavaScript los strings son como una colección de caracteres.

Por esta razón disponemos de propiedades y métodos muy útiles a la hora de trabajar con la información que hay adentro.

Los strings en JavaScript

En muchos sentidos, para JavaScript, un string no es más que un array de caracteres. Al igual que en los arrays, la primera posición siempre será 0.

```
{ } let nombre = 'Fran';
```

- **.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 un -1.

```
{  
  let saludo = '¡Hola! Estamos programando';  
  
  saludo.indexOf('Estamos'); // devuelve 7  
  saludo.indexOf('vamos'); // devuelve -1, no lo encontró  
  saludo.indexOf('o'); // encuentra la letra 'o' que está en la  
                        posición 2, devuelve 2 y corta la ejecución
```

A diferencia de las propiedades, llamamos métodos a las funciones que se encuentran dentro de objetos (los veremos en detalle en las próximas clases). A estos métodos debemos invocarlos como lo haríamos al llamar una función, con sus paréntesis y parámetros (si fuese necesario).

- **.slice()** : Corta el string y devuelve una parte del string donde se aplica.
 - Recibe 2 números como parámetros (pueden ser negativos):
 - El índice desde donde inicia el corte.
 - El índice hasta donde hacer el corte (es opcional).
 - Retorna la parte correspondiente al corte.

```
{  
  let frase = 'Breaking Bad Rules!';  
  
  frase.slice(9,12); // devuelve 'Bad'  
  frase.slice(13); // devuelve 'Rules!'  
  frase.slice(-10); // ¿Qué devuelve? ¡A investigar!
```

** Si se usa un índice negativo, indica el punto desde el final de la cadena. `string.slice(2, -1)` extrae desde tercer carácter hasta el último carácter de la cadena.*

- **.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'  
  
let nombreCompleto = '   Homero   J.   Simpson   ';  
nombreCompleto.trim(); // devuelve 'Homero   J.   Simpson'
```

- **.replace()** : Reemplaza una parte del string por otra.
 - Recibe dos strings como parámetros:
 - El string que queremos buscar.
 - El string que usaremos de reemplazo.
 - Retorna un nuevo string con el reemplazo.

```
{}  
  
let frase = 'Aguante Python!';  
frase.replace('Python', 'JS'); // devuelve 'Aguante JS!'  
frase.replace('Py', 'JS'); // devuelve 'Aguante JSthon!'
```

- **.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 cancion = 'And bingo was his name, oh!';  
  
cancion.split(' ');  
// devuelve ['And', 'bingo', 'was', 'his', 'name,', 'oh!']  
  
cancion.split(', ');  
// devuelve ['And bingo was his name', 'oh!']
```

length para saber la longitud

indexOf() para saber si existe algo

slice() para tomar una porción de texto

trim() para eliminar los espacios en blanco

split() para generar un array

replace() para cambiar una porción de texto

[Cheatsheet: Strings y Arrays](#)

Clase 11: Módulo II - Objetos literales

[Objetos literales](#)

En JavaScript, un objeto literal es una entidad independiente con propiedades. A su vez, esas propiedades tienen valores.

El concepto de objetos puede compararse con entidades de la vida real. Por ejemplo, un país representaría un objeto literal con ciertas propiedades: su nombre, cantidad de habitantes, capital, etc. Del mismo modo, los objetos literales en JavaScript pueden tener propiedades que definan sus características.

Métodos: Son funciones que se asignan como valor a una clave y que buscan hacer algo más allá de tener un simple dato. Al ser una función, necesita ser ejecutada.

Debemos poner los paréntesis después del nombre del método.

Estructura básica

Un objeto es una estructura de datos que puede contener propiedades y métodos.

Para crearlo usamos llave de apertura y de cierre {}.

```
{  
  let auto = {  
    patente : 'AC 134 DD',  
  };  
}
```

PROPIEDAD

Definimos el nombre de la **propiedad** del objeto.

DOS PUNTOS

Separa el nombre de la propiedad de su valor.

VALOR

Puede ser cualquier **tipo de dato** que conocemos.

Propiedades de un objeto

Un objeto puede tener la cantidad de propiedades que queramos. Si hay más de una, las separamos con comas ,.

Con la notación objeto.propiedad accedemos al valor de cada una de ellas.

```
{  
  let tenista = {  
    nombre: 'Roger',  
    apellido: 'Federer'  
  };  
  
  console.log(tenista.nombre) // Roger  
  console.log(tenista.apellido) // Federer  
}
```

Métodos de un objeto

Una propiedad puede almacenar cualquier tipo de dato. Si una propiedad almacena una función, diremos que es un método del objeto. Con una estructura similar a la de las funciones expresadas, vemos que se crean mediante el nombre del método, seguido de una función anónima.

```
{  
  let tenista = {  
    nombre: 'Roger',  
    edad: 38,  
    activo: true,  
    saludar: function() {  
      return '¡Hola! Me llamo Roger';  
    }  
  };  
}
```

Ejecución de un método de un objeto

Para ejecutar un método de un objeto usamos la notación `objeto.metodo()`. Los paréntesis del final son los que hacen que el método se ejecute.

```
{  
  let tenista = {  
    nombre: 'Roger',  
    apellido: 'Federer',  
    saludar: function() {  
      return '¡Hola! Me llamo Roger';  
    }  
  };  
  
  console.log(tenista.saludar()); // ¡Hola! Me llamo Roger  
}
```

Trabajando dentro del objeto

La palabra reservada **this** hace referencia al objeto en sí donde estamos parados. Es decir, el objeto en sí donde escribimos la palabra.

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;  
  }  
};  
console.log(tenista.saludar()); // ¡Hola! Me llamo Roger
```

Test

1. **Un objeto literal puede tener...** la cantidad de propiedades que queramos.
2. **Las propiedades de un objeto literal pueden almacenar...** cualquier tipo de dato.
3. **Seleccioná la sintaxis correcta para ejecutar el método de un objeto.**
objeto.metodo()
4. **Para separar las propiedades dentro de un objeto literal...** usamos el carácter coma ",".

Clase 12: Módulo II - Cierre de semana

Propiedad única:

```
console.log(persona['nombre']);
```