PROGRAMACIÓN CON LENGUAJES DE GUIÓN

CONCEPTOS PREVIOS

COMENTARIOS DE CÓDIGO

Si no has programado hasta ahora, debes conocer una serie de conceptos básicos que tendrás que trabajar y dominar dentro del campo de la programación.

Glosario general

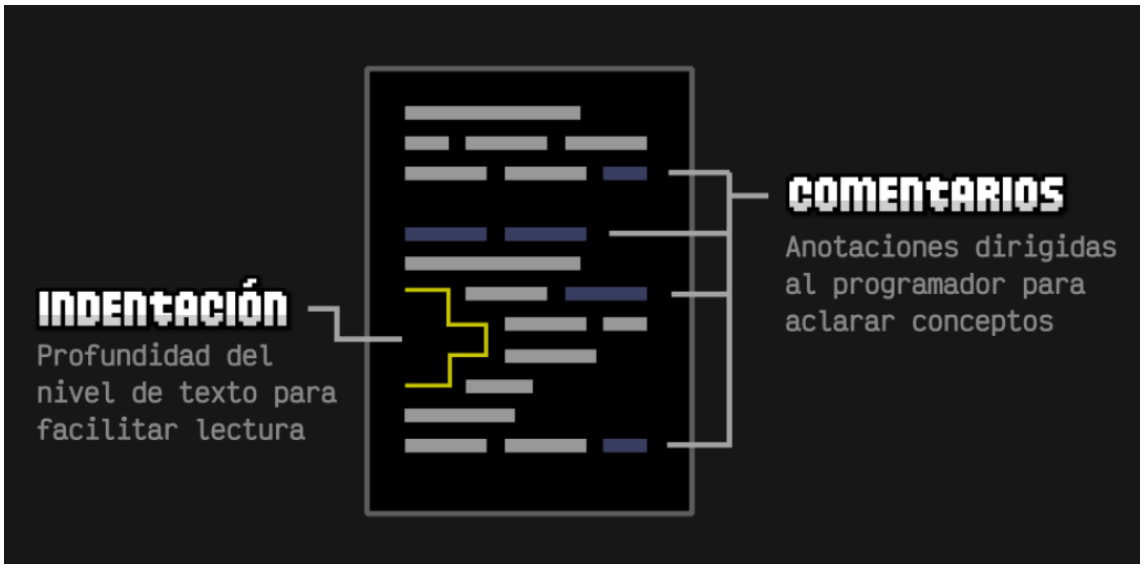
Programa: En programación se suele llamar «programa» a el conjunto total de código que desarrollas. En Desarrollo web, quizás el término más utilizado es sitio web (más orientado a contenido web) o aplicación web (más orientado a funcionalidad web). También se suele generalizar utilizando términos como «script» o «código Javascript», haciendo referencia a fragmentos más pequeños de esa aplicación.

Algoritmo: Un algoritmo es un conjunto de pasos conocidos, en un determinado orden, para conseguir realizar una tarea satisfactoriamente y lograr un objetivo.

Definición de Programa y algoritmo

Comentarios: Los comentarios en nuestro código son fragmentos de texto o anotaciones que el navegador ignora y no repercuten en el programa. Sirven para dejar por escrito detalles importantes para el programador. De esta forma cuando volvamos al código, nos será más rápido comprenderlo. Es una buena costumbre comentar en la medida de lo posible nuestro código.

Indentación: Se llama indentar a la acción de colocar espacios o tabuladores antes del código, para indicar si nos encontramos dentro de un if, de un bucle, etc... Esta práctica es muy importante y necesaria, y más adelante profundizaremos en ella.



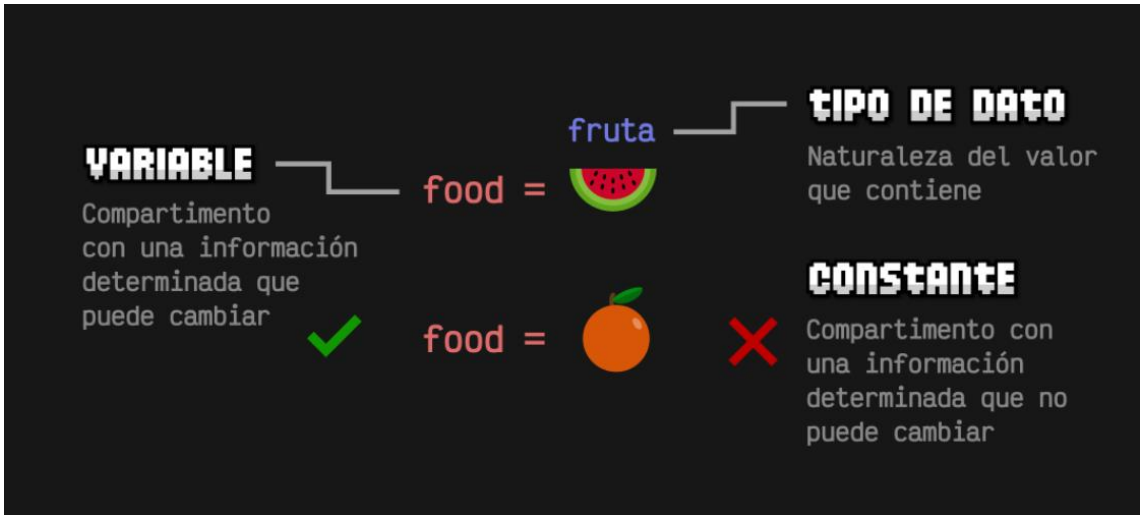
Definición de comentarios e indentación

Variables: Es el nombre que se le da a pequeños «compartimentos» (espacios en memoria) donde se guarda una información determinada, de forma muy similar a las incógnitas en matemáticas. Se llaman variables porque podemos modificarlas a lo largo del programa, según necesitemos. Un programa puede tener muchas variables, y cada una de ellas tendrá:

- **Un nombre** (para identificarlas y diferenciarlas de otras variables)
- **Un valor** (la información que contienen)
- **Un tipo de dato** (la naturaleza del valor que contiene).

```
x = 5; // nombre: x, valor: 5, tipo de dato: número
y = "Hola"; // nombre: y, valor: Hola, tipo de dato: texto
Manz = "me"; // nombre: Manz, valor: me, tipo de dato: texto
```

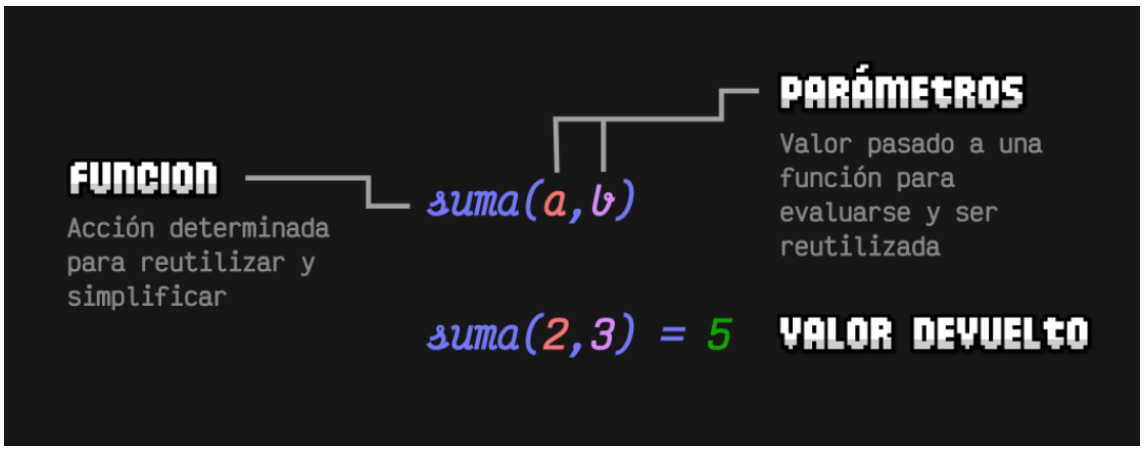
Constantes: Es el mismo concepto de una variable, salvo que, en este caso, la información que contiene será siempre constante (no puede variar).



Variables, constantes y tipos de datos

Funciones: Cuando comenzamos a programar, nuestro código se va haciendo cada vez más y más grande, por lo que hay que buscar formas de organizarlo y mantenerlo lo más simple posible. Las funciones son agrupaciones de código que, entre otras cosas, evitan que tengamos que escribir varias veces lo mismo en nuestro código. Una función contendrá una o mas acciones a realizar y cada vez que ejecutemos una función, se realizarán todas ellas.

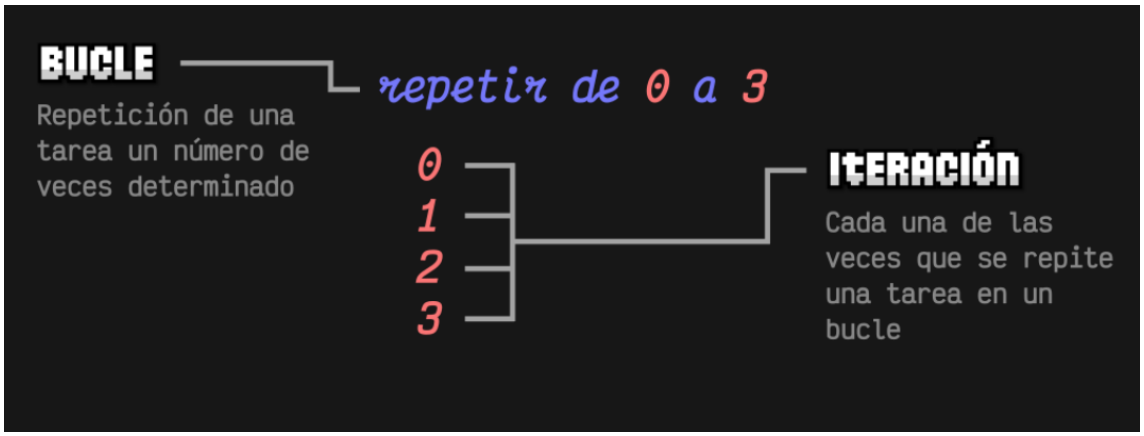
Parámetros: Es el nombre que reciben las variables que se le pasan a las funciones. Muchas veces también se les denomina argumentos.



Funciones y parámetros

Bucles: Cuando estamos programando, muchas veces necesitaremos realizar tareas repetitivas. Una de las ventajas de la programación es que permite automatizar acciones y no es necesario hacerlas varias veces. Los bucles permiten indicar el número de veces que se repetirá una acción. De esta forma, sólo la escribimos una vez en nuestro código, y simplemente indicamos el número de veces que queremos que se repita.

Iteración: Cuando el programa está en un bucle repitiendo varias veces la misma tarea, cada una de esas repeticiones se denomina iteración.



Bucles e iteraciones

Librería: Muchas veces, desarrollamos código que resuelve tareas o problemas que, posteriormente, querremos reutilizar en otros programas.

Cuando eso ocurre, en **Javascript** (y en otros lenguajes de programación) se suele empaquetar el código en lo que se llaman librerías, que no es más que código listo para que otros programadores puedan utilizarlo fácilmente en sus programas y beneficiarse de las tareas que resuelven de forma rápida y sencilla.

Comentarios de código

Cuando comenzamos a programar, por lo general, se nos suele decir que es una buena práctica mantener comentado nuestro código con anotaciones que faciliten la comprensión de las tareas que realizamos y los problemas que pretendemos solucionar, ya que el código que creamos no suele ser muy bueno, ni mucho menos descriptivo, puesto que estamos en fase de aprendizaje.

A medida que conseguimos destreza programando, notaremos que los comentarios son cada vez más prescindibles, sin embargo, conviene no dejar de comentar, sino en su lugar, aprender a comentar mejor.

Una serie de consejos a tener presentes a la hora de dejar comentarios en nuestro código:

- No comentes detalles redundantes. No escribas lo que haces, escribe por qué lo haces.
- Mejor nombrar variables/funciones/clases de forma descriptiva que usar comentarios para describirlas.
- Sé conciso y concreto. Resume. No escribas párrafos si no es absolutamente necesario.
- Intenta usar siempre el mismo idioma y estilo de comentarios.
- Si modificas código, revisa también los comentarios. Comentarios desactualizados, son inservibles.

Tipos de comentarios

En **Javascript** existen dos tipos de comentarios: **los comentarios de una sola línea y los comentarios de múltiples líneas.**

Una línea: Comienza con // y sólo comenta la línea actual desde donde se escribe.

Múltiples líneas: Comentarios extensos. Comienzan por /* y comentará todo el texto que escribamos hasta que cerremos el comentario con un */.

Veamos un ejemplo:

```
// Comentarios cortos de una sola línea. Suelen explicar la línea siguiente.
let a = 1;

let x = 45; // También se utilizan al final de una línea.

/* Por otro lado, existen los comentarios múltiples de varias líneas consecutivas.
   Suelen utilizarse para explicaciones largas que requieren bastante
   espacio porque se mencionan gran cantidad de cosas :-) */
```

Ejemplos

Comentar código también es un arte que debe ser aprendido, ya que al principio es muy fácil cometer errores y comentar en exceso o no ser concreto al comentar.

No suele ser grave porque los comentarios no afectan al funcionamiento del programa, pero en equipos de trabajo donde hay varios programadores suele ser molesto para los programadores con más experiencia.

Un ejemplo de comentario que suele ser contraproducente es aquel que se limita a decir lo que hacemos en la línea siguiente:

```
// Declaramos una variable llamada x ❌
let x = 50;

// La mostramos por consola ❌
console.log(x);

// Cambiamos su valor multiplicando por 0,5 ❌
x = x * 0.5;
```

Estos comentarios pueden ser útiles para el programador novato que comienza a programar y necesita recordar lo que hace porque aún no conoce bien la sintaxis de programación, de hecho muchos de los comentarios del tema de introducción son así (para ayudar al programador que recién empieza a programar), pero el objetivo real de un comentario no debe ser recordar que hace una línea de código, sino conocer porque lo estamos realizando o que representa lo que estamos haciendo:

```
let x = 50; // Establecemos el precio del producto ❌

console.log(x);

x = x * 0.5; // Lo rebajamos al 50% ❌
```

Sin embargo, hay una opción todavía mejor que conecta con uno de los temas que veremos más adelante. Poner nombres descriptivos a las variables debería ser algo obligatorio a lo que acostumbrarnos, puesto que puede ahorrarnos muchos comentarios y tiempo, simplificar el código considerablemente y hacerlo mucho más legible y con menos ambigüedades:

```
let precio = 50; ✅

console.log(precio);

let oferta = precio * 0.5; ✅
```

En este fragmento de código, no utilizamos comentarios porque el nombre de las variables ya ayuda a entender el código y lo hace autoexplicativo.

De esta forma, generamos menos código (e incluso comentarios) y se entiende igualmente. En los siguientes temas, veremos una serie de consejos a la hora de nombrar variables, funciones u otros elementos dentro de la programación.

Indentación de código

A medida que escribimos líneas de código en nuestro programa, este se irá complicando y nos tomará más tiempo leer lo que hemos hecho y comprobar si hay errores o buscar cómo solucionarlos.

Sin embargo, para mejorar la rapidez con la que leemos (y entendemos) el código, una buena práctica es usar la **indentación**.

¿Qué es la indentación?

Se llama indentación de código al hecho de utilizar sangrado (mover ligeramente hacia la derecha) las líneas de código, facilitando así la lectura, e indicando visualmente si nos encontramos en el interior de una estructura concreta, en el programa principal, etc...

Observemos el siguiente ejemplo (el código ahora no es importante, simplemente observa los espacios iniciales):

```
function action() {
  for (i = 0; i < 10; i++) {
    console.log("Iteración #", i);

    if (i == 9) {
      console.log("Estoy en la última iteración");
    } // if
  } // for
} // function
```

En el ejemplo anterior se puede observar que ciertas líneas tienen el texto indentado, incluso a diferentes niveles.

Los programadores **indentamos** el código para mostrar visualmente que fragmento de código actúa dentro de otro. De esta forma, es muy sencillo saber a qué nivel está actuando la línea de código en cuestión.

Por ejemplo, observando el ejemplo podemos ver rápidamente que el **for** actúa dentro de la **function**, y que el **if** está dentro del **for**, simplemente observando el nivel de indentación.

En algunos lenguajes de programación, como Python, la indentación es una característica obligatoria, que acostumbra al programador a indentar correctamente.

Es por esta razón, que Python se indica como un lenguaje ideal para adquirir buenos hábitos de programación.

Observemos este otro código (intencionadamente mal indentado):

```
function action() { for (i = 0; i < 10; i++) {
  console.log("Iteración #", i);
  if (i == 9) console.log("Estoy en la última iteración"); }
}
```

En principio, puede parecer que este código aprovecha mejor el espacio, porque ocupa menos, pero es mucho menos legible que el ejemplo anterior debido a su falta de indentación, ya que no queda claro que instrucciones están dentro de otras.

Una buena práctica de programación inicial es aprender a indentar correctamente las líneas de código que escribamos. Muchos editores de texto incorporan complementos que permiten que, al guardar, se revise la indentación del código y se corrija automáticamente.

Una buena práctica a la hora de programar en Javascript es ayudarnos de un linter como ESLint: una herramienta que analiza nuestro código en tiempo real y nos alerta de posibles errores de escritura. No se centra en corregir errores de funcionamiento, sino en la coherencia y la sintaxis escrita.

¿Tabuladores o espacios?

A la hora de indentar código hay dos aproximaciones popularmente extendidas: usar espacios o usar tabuladores. Utilizar una u otra estrategia de tabulación depende del programador, pero lo importante es ser coherente y siempre utilizar la misma.

Espacios: Si decidimos utilizar espacios, ten en cuenta que puedes elegir indentar con 2, 3 ó 4 espacios (por ejemplo). En los ejemplos de esta página suelo utilizar indentación a 2 espacios.

Tabuladores: Si decidimos utilizar tabuladores, debes saber que el carácter utilizado por la tecla TAB no es el mismo que el de los espacios.

Por ejemplo, un tabulador puede ser visualmente equivalente a 3 espacios, pero sólo ocupará un carácter. Muchos editores convierten automáticamente un tabulador a un número de espacios concreto.

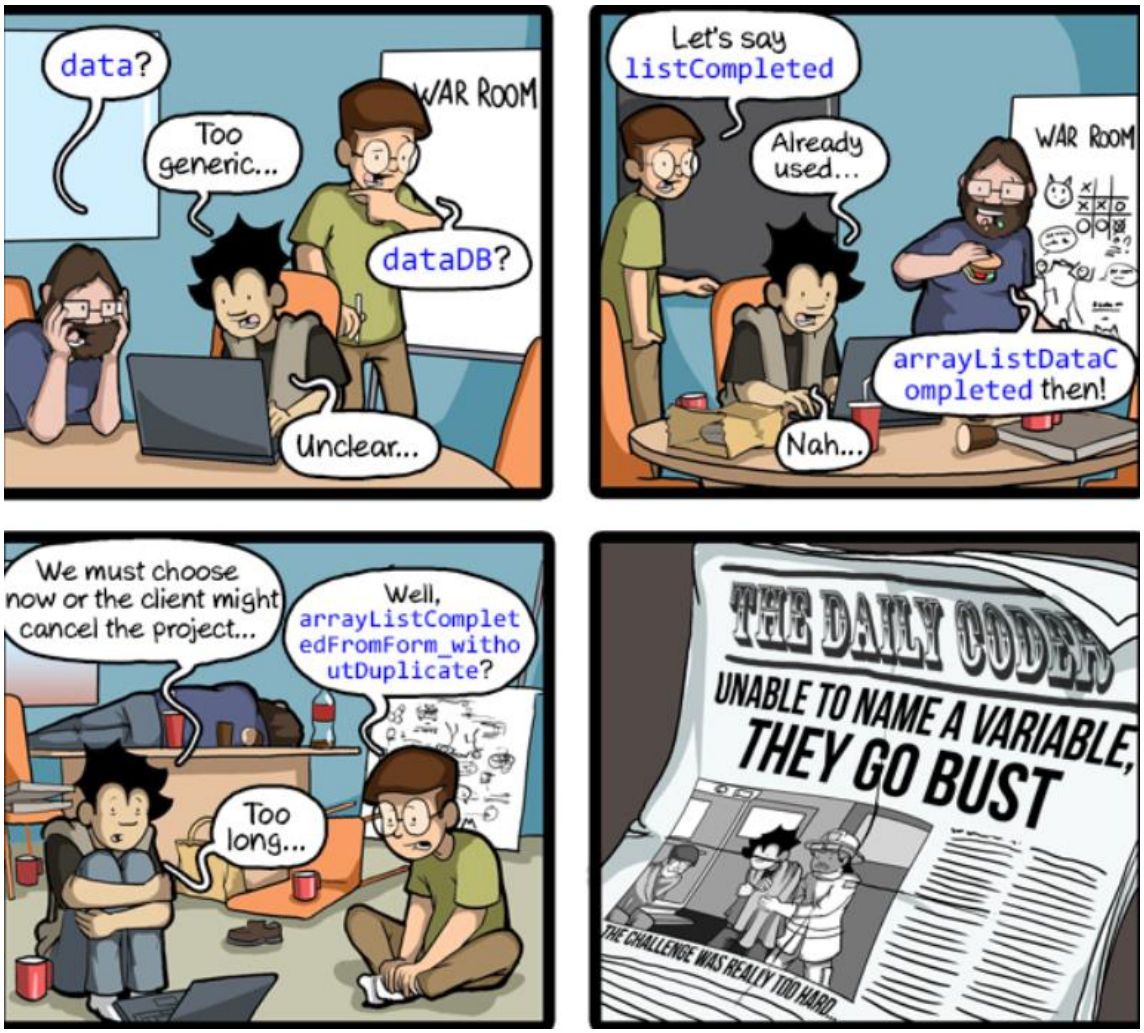
> En **VSCode** puedes activar la visualización de esta característica mediante puntos · (*espacios*) o flechas -> (*tabulador*), pulsando la tecla F1 y buscando/activando la opción «Alternar representación de espacio en blanco» (*Toggle Render Whitespace*)

Convenciones de nombres

Una de las tareas a priori más sencilla, pero a la misma vez de las más complejas, es la de ponerle un buen nombre a una variable (o a cualquier otro elemento de programación que necesite un nombre).

Los programadores necesitan utilizar variables en sus códigos, y además que estas variables tengan nombres que representen claramente la información que contienen.

Dejar nombres bien claros es muy importante para nosotros, ya que si necesitamos volver a trabajar con dicho código (o hacer modificaciones) nos resultará mucho más fácil. Sin embargo, esto cobra aún mayor importancia si otras personas tienen que revisar o modificar nuestro código, ya que ellos no están tan familiarizados con nuestro código y les ayudará mucho a tardar menos tiempo en comprenderlo.



Viñeta de Commit Strip sobre el nombrado de variables.

Veamos una serie de consejos y recomendaciones a la hora de establecer nombres en nuestro código.

La mayoría de ellas son simplemente convenciones, no son obligatorias para que el código funcione, pero son altamente recomendables para mejorar la calidad de nuestro código:

Nombres descriptivos

Una mala costumbre habitual cuando se empieza en la programación, es darle un nombre muy poco descriptivo de la información que contiene, simplemente porque es más corto y manejable.

Al principio, cuando tenemos pocas variables esto no importa demasiado, pero a medida que el código crece, se vuelve insostenible.

Es muy común tener que volver hacia atrás en nuestro código a cambiar nombres de variables porque hemos cambiado de parecer o porque se nos ha ocurrido un nombre mejor. Esto seguirá ocurriendo hasta que adquirimos cierta experiencia.

Evita nombres poco claros o inconsistentes como tmp, a, b2, variable2, etc...

Índices y contadores

A lo anterior, hay una pequeña excepción**. Cuando trabajamos en bucles for (o bucles en general), donde el ámbito de una variable que actúa como contador** (índice) es muy reducido (esa variable solo existe y afecta al interior del bucle), se suelen utilizar nombres de variables cortos para ser más productivo y claro.

Las variables que actúan como contador suelen nombrarse con una letra minúscula empezando desde i (de índice): i, j, k... A veces, también se usan letras como a, b, c... o la inicial minúscula de lo que representan: c para un contador, p para una posición, etc...

Constantes, clases y variables

Las **constantes** son variables especiales que no varían su valor a lo largo del programa y permanecen como su propio nombre indica: **constantes**. La convención adoptada con las constantes es que deben ir siempre en **MAYÚSCULAS** si se tratan de valores sensibles a ser modificados por el programador a lo largo del desarrollo del programa (**por ejemplo, una constante con los minutos con los que arrancará una cuenta atrás**).

Las **clases** son estructuras de código más complejas que veremos más adelante. Cuando necesites nombrarlas, es importante recordar que los nombres de las clases se escriben siempre capitalizadas: mayúsculas la primera letra y el resto en minúsculas.

Las **variables**, por último, siempre deben empezar por letra minúscula.

Independientemente de que sea **variable, constante o clase, su nombre nunca podrá empezar por un número**, sino que debe empezar por una letra o carácter. Si lo compruebas, verás que es imposible nombrar una variable que empiece por número.

Estilo de nombrado

Al margen del nombre que utilicemos para nombrar una variable, función u otro elemento, tenemos el estilo o convención que utilizaremos para escribir nombres compuestos por varias palabras.

Existen varias convenciones a utilizar:

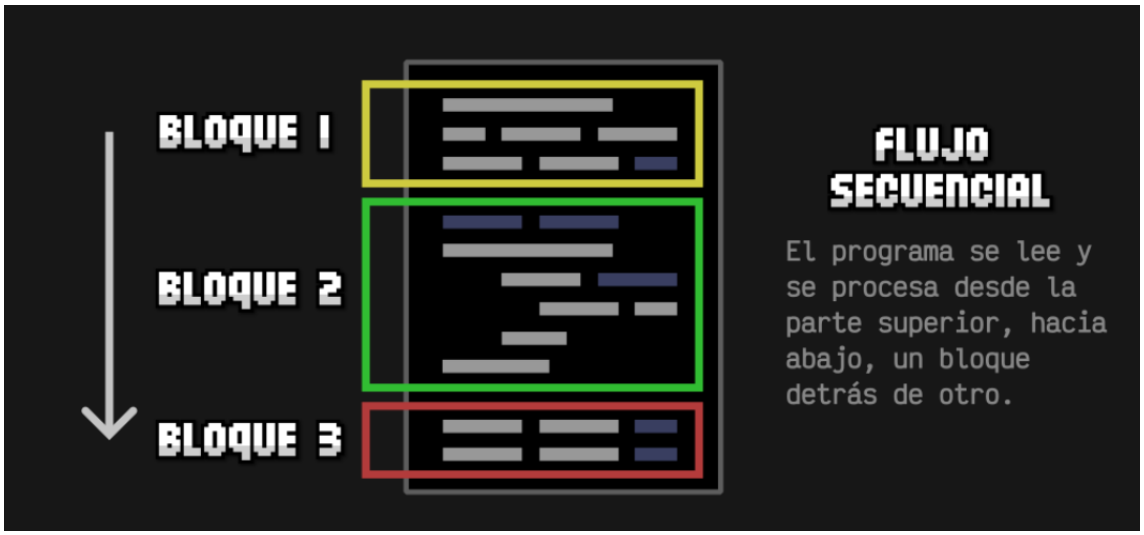
Nombre	Descripción	Ejemplo
lowercase	Todo minúsculas. ❌ No se usa porque en palabras compuestas puede confundir.	`precioproducto`
UPPERCASE	Todo mayúsculas. ❌ No se usa porque en palabras compuestas puede confundir.	`PRECIOPRODUCTO`
camelCase	Primera palabra, todo minúsculas. El resto, todo minúsculas salvo primera letra.	`precioProducto`
PascalCase	Idem, pero las palabras empiezan con la primera letra mayúscula. Se usa en Clases.	`PrecioProducto`
snake_case	Las palabras se separan con un guión bajo y se escriben siempre en minúsculas.	`precio_producto`
UPPER_SNAKE_CASE	Se usa en constantes ajustables frecuentemente por el programador.	`PRECIO_PRODUCTO`
kebab-case	Palabras separadas con un guión normal y en minúsculas. Usadas en HTML/CSS.	`precio-producto`
dot.case	Palabras en minúsculas separadas por puntos. ❌ En Javascript **no** se puede usar.	`precio.producto`
Húngara	Prefijo (minúsculas) con el tipo de dato (n = número, t = texto, ...). ❌ Desaconsejada.	`nPrecioProducto`

Una buena costumbre es ser consistente y no mezclarlos. Por ejemplo, en HTML la convención de nombrado para clases es kebab-case, mientras que en Javascript para variables se usa camelCase. Sin embargo, para clases, por ejemplo, se usa PascalCase.

Flujo de ejecución

En nuestro código de programación, en líneas generales, el flujo del programa, se lee desde arriba hacia abajo, y de izquierda a derecha.

De esta forma, y siempre por norma general, se procesa primero el código que está a la izquierda y arriba, actuando de forma secuencial, ya que cuando termina un primer bloque o línea de código, se comienza con el siguiente.



Flujo secuencial

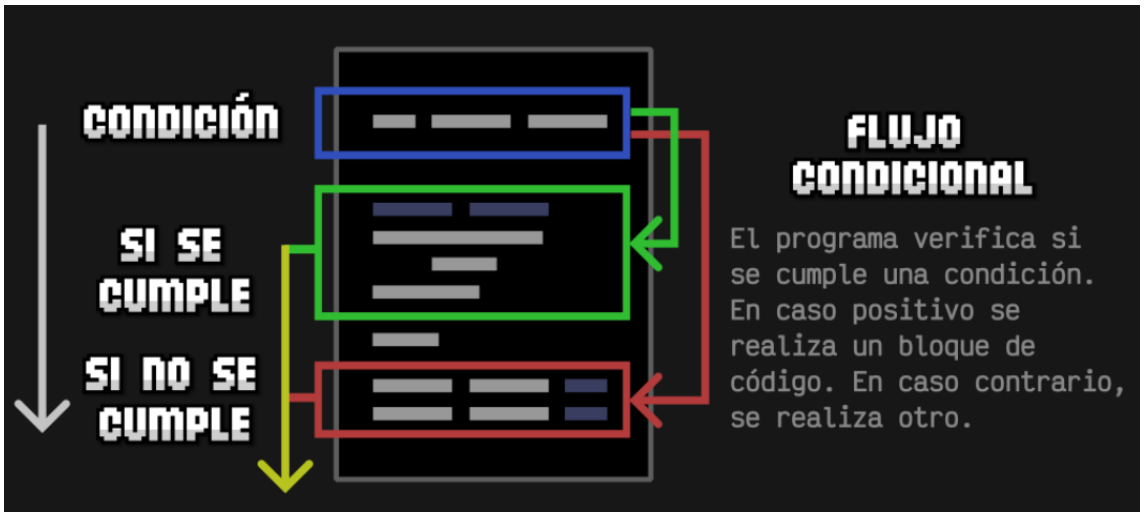
Este mecanismo es de los más sencillos y básicos, y a medida que vamos escribiendo código se puede complicar, o introducir variaciones que hacen que este flujo se vuelva más complejo.

Nombre	Descripción
Condiciones	Bifurcaciones donde el flujo se divide en dos o más caminos.
Bucles	Repeticiones de un código idéntico varias veces o hasta cumplir una condición.
Funciones	Fragmento de código que realiza una tarea, abstrayendo información.
Anidación	Fragmentos o bloques de código dentro de otros.
Estructuras de datos	Lugares o «compartimentos» donde podemos guardar información.

Condicionales

Los condicionales, condiciones o bifurcaciones son fragmentos de código donde se establece una cierta condición para evaluar si realizar un bloque de código u otro.

De esta forma, tenemos un bloque de código (en verde) que sólo se realizará si se cumple la condición, saltando el bloque rojo y continuando el programa, o un bloque de código (en rojo) que sólo se realiza si no se cumple la condición, saltando el bloque verde y continuando el programa:



Flujo condicional

Las condiciones pueden ser algo más complejas, o incluso tener expresiones que deben ser evaluadas. Las explicaremos más adelante.

Bucles

Los bucles, ciclos o estructuras de repetición, son bloques especiales de código que se ejecutarán varias veces (de 0 a un número concreto de veces) dependiendo de una condición. Esto nos ahorra el tener que repetir código muchas veces, y hace que el programa sea más pequeño y más fácil de leer y escribir.

El flujo de un bucle es el siguiente:

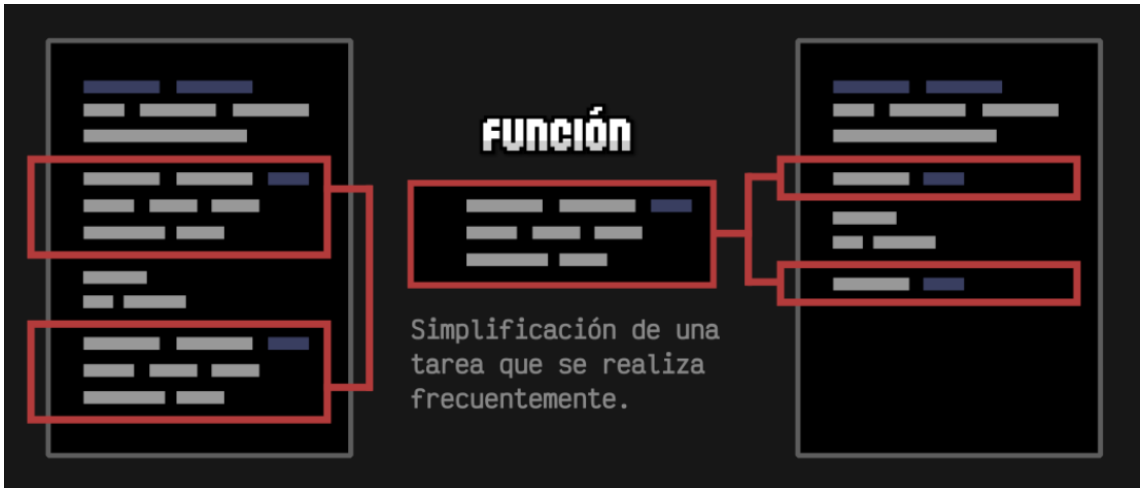
- Evaluar y determinar si una condición es cierta o falsa
- Si es cierta, ejecuta el bloque de código, realiza un cambio relacionado con la condición y vuelve a 1.
- Si es falsa, sale del bucle y continua el programa.

Cada una de las repeticiones que realiza es un concepto que se denomina iteración o vuelta.

Funciones

A medida que escribimos código en nuestra aplicación, es muy común que nos encontremos con que hay partes del código que se repiten y/o se parecen mucho a otras (o directamente son idénticas).

Existe un concepto llamado función que básicamente lo que permite es aislar un fragmento de código y darle un nombre. De esta forma, cada vez que llamemos a ese nombre, se aplicará el fragmento de código. Esto reduce sustancialmente el código de nuestra aplicación y lo hace mucho más fácil de leer y entender.



Las funciones son mucho más complejas, pero esto es un buen punto de partida para entenderlas. Más adelante las explicaremos en profundidad.

Anidación

Otro concepto que conviene conocer es el denominado **anidación o anidamiento**.

Cuando hablamos de un **código anidado** **significa que tenemos un fragmento de código «dentro» de otro**.

Por ejemplo, imaginemos que tenemos un condicional que, a su vez, tiene dentro otro condicional. En ese caso estaríamos hablando un condicional anidado.

Si tenemos un bucle con otro bucle en su interior, estaríamos hablando de un bucle anidado.

También es interesante conocer el nivel de anidamiento, ya que en los casos mencionados sólo hay 2 niveles, pero podríamos hablar de niveles superiores de anidación donde por ejemplo tenemos 3 condicionales anidados o más.

Aunque ya lo veremos más adelante, hay que tener cuidado con las anidaciones, ya que pueden complicar nuestro programa, y en algunos casos hacerlo menos eficiente. Lo hablaremos en profundidad en próximos temas.

Estructuras de datos

Por último, y no por ello menos importante, tenemos las denominadas **estructuras de datos**.

Cuando programamos, de forma habitual necesitamos guardar información en nuestro programa, y a priori no conocemos dicha información porque variará dependiendo del usuario.

Por ejemplo, imagina un programa que pregunta el nombre al usuario. En un caso podría ser Juan, en otro caso podría ser Sara, y en otro podría ser Pancracio.

Es por ello, que necesitamos utilizar ciertas estructuras de datos (compartimentos) para almacenar dicha información y usarla en nuestro programa de forma abstracta o genérica.

Estos compartimentos se identificarán en nuestro programa mediante un nombre identificativo, sin mencionar el valor que contienen directamente.

Más adelante veremos los diferentes tipos de estructuras de datos que existen, ya que algunos se adaptan mejor que otros para almacenar la información, dependiendo de lo que necesitemos.

Condicionales If / else

Cuando escribimos código, por lo general, se lee de forma secuencial, es decir, una línea detrás de otra, desde arriba hacia abajo.

Por lo tanto, una acción que realicemos en la línea 5 nunca ocurrirá antes que una que aparece en la línea 3.

Ya veremos que más adelante esto se complica, pero en principio partimos de esa base, como explicamos en el flujo de ejecución de un programa.

Condicionales

Al hacer un programa necesitaremos establecer condiciones o decisiones, donde buscamos que se realice una acción A si se cumple una condición o una acción B si no se cumple.

Este es el primer tipo de **estructuras de control** que encontraremos.

Tenemos varias estructuras de control condicionales:

Estructura de control	Descripción
`If`	**Condición simple**: Si ocurre algo, haz lo siguiente...
`If/else`	**Condición con alternativa**: Si ocurre algo, haz esto, sino, haz esto otro...
`?:`	**Operador ternario**: Equivalente a `If/else`, forma abreviada.
`Switch`	**Estructura para casos específicos**: Similar a varios `If/else` anidados.

Condicional If

Quizás, el más conocido de estos mecanismos de estructura de control es el if (condicional). Con él podemos indicar en el programa que se tome un camino sólo si se cumple la condición que establezcamos.

Observa el siguiente ejemplo, donde guardamos en el «compartimento» nota, un valor numérico:

```
let nota = 7;
console.log("He realizado mi examen.");

// Condición (si la nota es mayor o igual a 5)
if (nota ⩾ 5) {
  console.log("¡Estoy aprobado!");
}
```

En este caso, como el valor de nota es superior o igual a 5, nos aparecerá en la consola el mensaje «¡Estoy aprobado!». Sin embargo, si modificamos en la primera línea el valor de nota a un valor inferior a 5, no nos aparecerá ese mensaje.

Cuando dentro de las llaves ({ }) sólo tenemos una línea, se pueden omitir dichas llaves. Aun así, es recomendable ponerlas siempre si tenemos dudas o no estamos seguros.

Condicional If / else

Se puede dar el caso que queramos establecer una alternativa a una condición. Para eso utilizamos el if seguido de un else.

Con esto podemos establecer una acción A si se cumple la condición, y una acción B si no se cumple.

Vamos a modificar el ejemplo anterior para mostrar también un mensaje cuando estamos suspendidos, pero en este caso, en lugar de mostrar el mensaje directamente con un console.log vamos a guardar ese texto en una nueva variable calificacion:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A: nota es menor que 5
  calificacion = "suspendido";
} else {
  // Acción B: Cualquier otro caso diferente a A (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

Nuevamente, en este ejemplo comprobaremos que podemos conseguir que se muestre el mensaje Estoy aprobado o Estoy suspendido dependiendo del valor que tenga la variable nota.

La diferencia con el ejemplo anterior es que creamos una nueva variable que contendrá un valor determinado dependiendo de la condición del If.

Por último, el **console.log del final**, muestra el contenido de la variable calificacion, independientemente de que sea el primer caso o el segundo.

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A (nota es menor que 5)
  calificacion = "suspendido";
}
if (nota >= 5) {
  // Acción B (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

Este nuevo ejemplo, es equivalente al ejemplo anterior. Si nos fijamos bien, **la única diferencia respecto al anterior es que estamos realizando dos if independientes**: uno para comprobar si está suspendido y otro para comprobar si está aprobado.

Pero, aunque son equivalentes, no son exactamente iguales, ya que en el ejemplo que vimos anteriormente sólo existe un if, y por lo tanto, sólo se realiza la comprobación de una condición.

En este ejemplo que vemos ahora, se realizan dos if, y por lo tanto, se evalúan dos condiciones independientes.

En este caso se trata de algo insignificante que no repercute demasiado, pero es importante darse cuenta de que el primer ejemplo estaría realizando menos tareas para conseguir un mismo resultado.

Condicional If múltiple

Es posible que necesitemos crear un condicional múltiple con más de 2 condiciones, por ejemplo, para establecer la calificación específica.

Para ello, podemos anidar varios if/else uno dentro de otro, de la siguiente forma:

```
let nota = 7;
console.log("He realizado mi examen.");

// Condición
if (nota < 5) {
  calificacion = "Insuficiente";
} else if (nota < 6) {
  calificación = "Suficiente";
} else if (nota < 8) {
  calificacion = "Bien";
} else if (nota ⩽ 9) {
  calificacion = "Notable";
} else {
  calificacion = "Sobresaliente";
}

console.log("He obtenido un", calificacion);
```

Sin embargo, anidar de esta forma varios if suele ser muy poco legible y produce un código repetitivo algo feo.

En algunos casos se podría utilizar otra estructura de control llamada switch, que veremos a continuación.

Condicionales Switch

Si has visto la sección de condicionales If/else, habrás notado que su último ejemplo, donde utilizamos múltiples condiciones if / else no es demasiado elegante, ya que existe mucho código repetitivo. Si tenemos la necesidad de hacer múltiples condiciones y queremos solucionar esto, podemos recurrir al switch.

Condicional Switch

La estructura de control switch permite definir casos específicos a realizar cuando la variable expuesta como condición sea igual a los valores que se especifican a continuación mediante cada case:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

switch (nota) {
  case 10:
    calificacion = "Sobresaliente";
    break;
  case 9:
  case 8:
    calificacion = "Notable";
    break;
  case 7:
  case 6:
    calificacion = "Bien";
    break;
  case 5:
    calificacion = "Suficiente";
    break;
  case 4:
  case 3:
  case 2:
  case 1:
  case 0:
    calificacion = "Insuficiente";
    break;
  default:
    // Cualquier otro caso
    calificacion = "Nota errónea";
    break;
}

console.log("He obtenido un", calificacion);
```

Hay varias puntualizaciones que aclarar sobre este ejemplo, así que vamos a explicarlo:

- **La sentencia switch establece que vamos a realizar múltiples condiciones analizando la variable nota.**
- **Cada condición se establece mediante un case, seguido del valor posible de cada caso**.
- El switch comienza evaluando el primer case, y continua con el resto, hacia abajo.
- **Observa que algunos case tienen un break. Esto hace que deje de evaluar y se salga del switch**.
- **Los case que no tienen break, no se interrumpen, sino que se salta al siguiente case.**
- El caso especial default es como un else. Si no entra en ninguno de los anteriores, entra en default.

Ten en cuenta que este ejemplo no es exactamente equivalente al que vimos en el tema de if/else. Este ejemplo funcionaría si sólo permitimos notas que sean números enteros, es decir, números del 0 al 10, sin decimales. En el caso de que nota tuviera por ejemplo, el valor 7.5, mostraría Nota errónea, ya que no entraría en los casos anteriores.

El ejemplo de los if múltiples del tema anterior si controla casos de números decimales porque establecemos comparaciones de rangos con mayor o menor (> ó <), **cosa que con el switch no se puede hacer**. **El switch está indicado para casos con valores concretos y específicos.**

Recuerda que al final de cada caso es necesario indicar un break para terminar y salir del switch. En el caso que no sea haga, el programa saltará al siguiente caso, incluso aunque no se cumpla la condición específica.

Alternativas al Switch

Aunque puede funcionar perfectamente, a muchos programadores el switch no les resulta una estructura condicional satisfactoria, ya que tiene una sintaxis algo extraña que puede complicarse, puede ser difícil de leer y poco elegante. Existen algunas alternativas que veremos más adelante, como el uso de diccionarios y objetos, o incluso el uso de operadores ternarios, que muchas veces, también son desaconsejados en casos con muchas condiciones.

Operador ternario

Si has visto el tema donde hablamos de los condicionales if/else, habrás visto la forma más habitual de establecer condiciones en un programa.

Sin embargo, existe una forma alternativa y compacta de escribir una condición, haciendo uso del operador ternario.

Operador ternario

El operador ternario es una alternativa al condicional if/else de una forma mucho más compacta y breve, que en muchos casos resulta más legible. Sin embargo, hay que tener cuidado, porque su sobreutilización puede ser contraproducente y producir un código más difícil de leer.

La sintaxis de un operador ternario es la siguiente:

```
condición ? valor verdadero : valor falso;
```

Para entenderlo bien, vamos a reescribir el ejemplo de los temas anteriores utilizando este operador ternario. Primero, recordemos el ejemplo utilizando estructuras if/else:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

if (nota < 5) {
  // Acción A: nota es menor que 5
  calificacion = "suspendido";
} else {
  // Acción B: Cualquier otro caso diferente a A (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

Ahora, vamos a reescribirlo utilizando un operador ternario:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Operador ternario: (condición ? verdadero : falso)
let calificacion = nota < 5 ? "suspendido" : "aprobado";

console.log("Estoy", calificacion);
```

Repasemos el ejemplo:

Observa que guardamos en calificacion el resultado del operador ternario.

- **La condición es nota < 5, se escribe al principio, previo al ?.**
- Si la condición es cierta, el ternario devuelve "suspendido".
- Si la condición es falsa, el ternario devuelve "aprobado".

Este ejemplo hace exactamente lo mismo que el ejemplo anterior del if/else.

La idea del operador ternario es que podemos condensar mucho código y tener un if en una sola línea.

Es muy práctico, legible e ideal para ejemplos pequeños donde almacenamos la información en una variable para luego utilizarla.

Operador ternario anidado

Sin embargo, hay que tener cuidado, ya que los operadores ternarios sólo se recomiendan cuando se trata de if muy pequeños.

Si intentásemos realizar una comprobación de if múltiples con el operador ternario, la sintaxis puede resultar compleja y difícil de leer.

Observa el siguiente ejemplo con múltiples if / else:

```
let nota = 7;
console.log("He realizado mi examen.");

if (nota < 5) {
  calificacion = "Insuficiente";
} else if (nota < 6) {
  calificación = "Suficiente";
} else if (nota < 8) {
  calificacion = "Bien";
} else if (nota ⩽ 9) {
  calificacion = "Notable";
} else {
  calificacion = "Sobresaliente";
}

console.log("He obtenido un", calificacion);
```

Vamos a trasladarlo a un nuevo ejemplo utilizando operadores ternarios anidados:

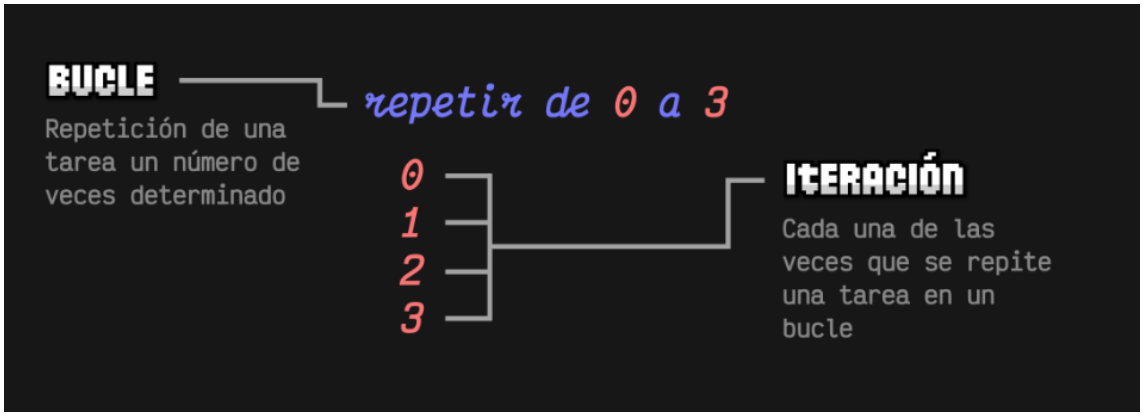
```
let nota = 7;
console.log("He realizado mi examen.");

let calificacion =
  nota < 5 ? "Insuficiente" :
  nota < 6 ? "Suficiente" :
  nota < 8 ? "Bien" :
  nota ⩽ 9 ? "Notable" :
  "Sobresaliente";

console.log("He obtenido un", calificacion);
```

Observa que lo que tenemos en este ejemplo son múltiples operadores ternarios anidados uno dentro de otro. El "valor falso" del primer operador ternario, es un nuevo operador ternario, que a su vez su valor falso es un nuevo operador ternario, y así con varios casos más.

Aunque en principio puede resultar interesante porque es bastante compacto y se escribe poco código, se trata de una estructura y sintaxis extraña, por lo que recuerda que los operadores ternarios anidados no suelen estar muy bien vistos.

¿Qué son los bucles?

Una de las principales ventajas de la programación es la posibilidad de crear bucles y repeticiones para tareas específicas, y que no tengamos que realizar el mismo código varias veces de forma manual.



Bucles e iteraciones

Los bucles nos permiten simplificar nuestro código, que sea más fácil de leer e incluso más fácil de modificar y mantener.

Tipos de bucles

Existen muchas formas de realizar bucles, vamos a ver los más básicos, que son muy similares en otros lenguajes de programación:

Tipo de bucle	Descripción
`while`	Bucles simples.
`for`	Bucles clásicos por excelencia.
`do..while`	Bucles simples que se realizan siempre como mínimo una vez.
`for..in`	Bucles sobre posiciones de un array. Los veremos más adelante.
`for..of`	Bucles sobre elementos de un array. Los veremos más adelante.
Array functions	Bucles específicos sobre arrays. Los veremos más adelante.

Los tres primeros son los bucles por excelencia en la programación. Un poco más adelante, cuando conozcamos ciertas estructuras de datos y funciones, también podremos ampliar y **conocer los bucles basados en arrays**, pero de momento nos centraremos en los primeros.

Conceptos de bucles

Antes de comenzar a aprender cómo funcionan los diferentes tipos de bucles existen, es necesario conocer algunos conceptos básicos relacionados con los bucles.

Condición

Al igual que en los condicionales if, en los bucles se va a evaluar una condición para saber si se debe seguir repitiendo el bucle o se debe finalizar.

Habitualmente, lo que se suele hacer es establecer que, si la condición es verdadera, se vuelve a repetir el bucle.

Por el contrario, si es falsa, se finaliza. Sin embargo, esta condición puede variar dependiendo de la implementación que le indique el programador.

Iteración

Otro concepto que usaremos mucho dentro de un bucle es el concepto de Iteración. Esto se refiere a cada una de las repeticiones de un bucle. Por ejemplo, si un bucle se repite como en el gráfico superior de este artículo (de 0 a 3*), se dice que hay 4 iteraciones.

Mucho cuidado con los bucles, porque en programación se suele empezar a contar en 0, por lo que esa es la primera iteración. Si el bucle va desde 0 a 3, hay 4 iteraciones.

Contador

Muchas veces, los bucles que creamos incorporan un contador, que no es más que algo que irá guardando un número para contar el número de repeticiones realizadas, y así finalizar cuando se llegue a otro número concreto. Dicho contador hay que inicializarlo (crearlo y darle un valor) antes de comenzar el bucle.

Incremento

Al igual que tenemos un contador en un bucle, también debemos tener una parte donde hagamos un incremento (o un decremento) de dicho contador. Si no lo tuvieramos, el contador no cambiaría y la condición siempre sería veradera, por lo que sería imposible salir del bucle.

Bucle infinito

Cuando estamos aprendiendo a programar, es muy común que cometamos un error creando el bucle y nos quedemos en un bucle infinito, es decir, en una situación donde nuestro programa se queda eternamente en bucle y nunca termina. Como programadores, esta situación siempre hay que evitarla. Para ello, lo que debemos hacer es siempre comprobar que existe un incremento (o decremento) y que en algún momento la condición va a ser falsa y se podrá salir del bucle.

Ten en cuenta que de producirse un bucle infinito, nuestro programa se quedará atascado y tendremos que forzar para finalizarlo. Ten siempre cuidado al crear un bucle para que no sea infinito.

Bucles while

El primer tipo de bucle se denomina bucle while. Este bucle se caracteriza en que se repite, revisando la condición en cada iteración y sólo se detiene cuando la condición es falsa.

Bucle while

El bucle while es uno de los bucles más simples que podemos crear. Vamos a repasar el siguiente ejemplo y analizar todas sus partes, para luego analizar lo que ocurre en cada iteración del bucle. Empecemos por un fragmento sencillo del bucle:

```
let i = 0;  // Inicialización de la variable contador

// Condición: Mientras la variable contador sea menor de 5
while (i < 5) {
  console.log("Valor de i:", i);

  i = i + 1; // Incrementamos el valor de i
}
```

Veamos qué es lo que ocurre a la hora de ejecutar este código:

- Antes de entrar en el bucle while, se inicializa la variable i al valor 0.
- Antes de realizar la primera iteración del bucle, comprobamos la condición.
- Si la condición es verdadera, hacemos las tareas que están indentadas dentro del bucle.
- Mostramos por pantalla el valor de i.
- Luego, incrementamos el valor de i sumándole 1 a lo que ya teníamos en i.
- Terminamos la iteración del bucle, por lo que volvemos al inicio del while a hacer una nueva iteración.
- Volvemos al punto 2) donde comprobamos de nuevo la condición del bucle.
- Repetimos hasta que la condición sea falsa. Entonces, salimos del bucle y continuamos el programa.

Traza del bucle

Una tarea que suele servir para entender mejor el bucle, es hacer una traza de lo que haría un programa internamente. Esto es ir realizando cada paso, para entenderlo bien.

Veamos una muestra paso a paso de las iteraciones de este ejemplo anterior:

Iteración del bucle	Valor de i	Descripción	Incremento
Antes de empezar	i = undefined	Antes de comenzar el programa.	
Iteración #1	i = 0	¿(0 < 5)? Verdadero. Mostramos 0 por pantalla.	i = 0 + 1
Iteración #2	i = 1	¿(1 < 5)? Verdadero. Mostramos 1 por pantalla.	i = 1 + 1
Iteración #3	i = 2	¿(2 < 5)? Verdadero. Mostramos 2 por pantalla.	i = 2 + 1
Iteración #4	i = 3	¿(3 < 5)? Verdadero. Mostramos 3 por pantalla.	i = 3 + 1
Iteración #5	i = 4	¿(4 < 5)? Verdadero. Mostramos 4 por pantalla.	i = 4 + 1
Iteración #6	i = 5	¿(5 < 5)? Falso. Salimos del bucle.	

El bucle while es muy simple, pero requiere no olvidarse accidentalmente de la inicialización y el incremento (además de la condición). Más adelante veremos otro tipo de bucle denominado bucle for, que tiene una sintaxis diferente y, habitualmente, se suele utilizar más en el día a día.

La operación i = i + 1 es lo que se suele llamar un incremento de una variable. Es muy común simplificarla como i++, que hace exactamente lo mismo: aumenta en 1 su valor.

Bucle do ... while

Existe una variación del bucle while denominado bucle do while. La diferencia fundamental, aparte de variar un poco la sintaxis, es que este tipo de bucle siempre se ejecuta una vez, al contrario que el bucle while que en algún caso podría no ejecutarse nunca.

Para entenderlo bien, antes de ver el bucle do while vamos a analizar el siguiente ejemplo:

```
let i = 5;

while (i < 5) {
  console.log("Hola a todos");
  i = i + 1;
}

console.log("Bucle finalizado");
```

Observa, que, aun teniendo un bucle, este ejemplo nunca mostrará el texto Hola a todos, puesto que la condición nunca será verdadera, porque ya ha empezado como falsa (i ya vale 5 desde el inicio). Por lo tanto, nunca se llega a realizar el interior del bucle.

Con el bucle do while podemos obligar a que siempre se realice el interior del bucle al menos una vez:

```
let i = 5;

do {
  console.log("Hola a todos");
  i = i + 1;
} while (i < 5);

console.log("Bucle finalizado");
```

Observa los siguientes detalles de la variación do while:

- En lugar de utilizar un while desde el principio junto a la condición, escribimos do.
- El while con la condición se traslada al final del bucle.
- Lo que ocurre en este caso es que el interior del bucle se realiza siempre, y sólo se analiza la condición al terminar el bucle, por lo que, aunque no se cumpla, se va a realizar al menos una vez.

Este ejemplo suele ser interesante cuando queremos establecer un bucle, pero queremos que siempre se realice una vez, independientemente de si cumple o no la condición.

Bucles for

El segundo tipo de bucle que veremos se denomina bucle for.

Este bucle se caracteriza en que se va a repetir, revisando la condición en cada iteración, hasta que no se cumpla la condición propuesta.

Bucle for

El bucle for es quizás uno de los más utilizados en el mundo de la programación. En Javascript se utiliza exactamente igual que en otros lenguajes como Java o C/C++. Veamos un ejemplo muy similar al que hemos realizado con el bucle while en el tema anterior:

```
// for (inicialización; condición; incremento)
for (let i = 0; i < 5; i++) {
  console.log("Valor de i:", i);
}
```

Como vemos, la sintaxis de un bucle for es mucho más compacta y rápida de escribir que la de un bucle while, sin embargo puede parecernos más críptica cuando la vemos por primera vez.

La sintaxis del bucle for es mucho más práctica porque te obliga a escribir la inicialización, la condición y el incremento antes del propio bucle, y eso hace que no te olvides de estos tres puntos fundamentales, cosa que suele ocurrir en los bucles while, lo que suele desembocar en un bucle infinito. Aunque también puede ocurrir en el bucle for, suele ser menos habitual.

Analicemos la sintaxis del bucle:

- Separemos por; lo que establecemos dentro de los paréntesis del for.
- Lo primero es la inicialización let i = 0. Esto ocurre sólo una vez antes de empezar el bucle.
- Lo segundo es la condición i < 5. Esto se comprueba al principio de cada iteración.
- **Lo tercero es el incremento i++, es decir, i = i + 1. Esto ocurre al final de cada iteración.**

Si lo pensamos bien, es lo mismo que hacemos en el bucle while, pero escribiéndolo de otra forma:

```
let i = 0;
while (i < 5) {
  console.log("Valor de i:", i);
  i++;
}
```

Recuerda que en programación es muy habitual empezar a contar desde cero. Mientras que en la vida real se contaría desde 1 hasta 10, en programación se contaría desde 0 hasta 9.

Decremento

No nos acostumbremos a hacer los bucles de memoria, ya que las condiciones pueden variar y ser bien diferentes. Por ejemplo, vamos a hacer un bucle que, en lugar de incrementar su contador, se decremente, ya que nos interesa hacer una cuenta atrás:

```
for (let i = 5; i > 0; i--) {
  console.log("Valor de i:", i);
}
```

En este caso, vamos a arrancar el bucle con un valor de i de 5.

Repetiremos la iteración varias veces, y observa que el incremento que tenemos es en su lugar un decremento, por lo que en lugar de sumarle 1, lo restamos.

El valor de i iría desde 5, a 4, 3, 2, 1 y cuando se reduzca a 0, ya no cumpliría la condición i > 0, por lo que terminaría el bucle, saldría de él y continuaría el resto del programa.

Incremento múltiple

Aunque no suele ser habitual, es posible añadir varias inicializaciones o incrementos en un bucle for separando por comas.

En el siguiente ejemplo además de aumentar el valor de una variable i, inicializamos una variable j con el valor 5 y la vamos decrementando:

```
for (i = 0, j = 5; i < 5; i++, j--) {
  console.log("Valor de i y j:", i, j);
}
```

Este código sería equivalente al siguiente:

```
let i = 0;
let j = 5;

while (i < 5) {
  console.log("Valor de i: ", i);
  console.log("Valor de j: ", j);

  i++;
  j--;
}
```

En principio, el programador debería realizar el bucle con el que más cómodo se sienta, tanto un while como un for. Sin embargo, es más común encontrarse con bucles for en el día a día, por lo que se recomienda no dejar de utilizar el for simplemente por un rechazo inicial debido a que su sintaxis pueda parecer más compleja.

Interrumpir bucles

Por norma general, un buen hábito de programación cuando creamos bucles es pensar la forma de hacer que esos bucles siempre vayan desde un número inicial a un número final y terminen las repeticiones. De esta forma son predecibles y fáciles de leer. Sin embargo, en algunas ocasiones nos puede interesar hacer interrupciones o saltos de iteraciones para conseguir algo más específico que con un bucle íntegro es más complejo de conseguir.

Saltar una iteración

Imagina un caso donde queremos saltarnos una iteración concreta. Por ejemplo, queremos hacer un bucle que muestre los números del 0 al 10, pero queremos que el número 5 se lo salte y no lo muestre, continuando con el resto.

Para eso podemos hacer uso de continue, que es una sentencia que al llegar a ella dentro de un bucle, el programa salta y abandona esa iteración, volviendo al principio del bucle:

```
for (let i = 0; i < 11; i++) {
  if (i == 5) {
    continue;
  }

  console.log("Valor de i:", i);
}
```

Así pues, en este caso se van a realizar las iteraciones desde 0 hasta 4, en la iteración del 5 se entrará en el if y se evaluará como verdadera, ejecutando el continue y saltando a la siguiente iteración. Por lo tanto continuará desde la iteración 6 hasta la 10.

En resumen, la iteración 5 nunca llegará al console.log, por lo que no se mostrará por pantalla.

Alternativas

Este ejemplo es muy sencillo, pero nos sirve para ver cómo funciona continue de una forma simple. También se podría haber solucionado sin necesidad de utilizar continue de la siguiente forma:

```
for (let i = 0; i < 11; i++) {
  if (i != 5) {
    console.log("Valor de i:", i);
  }
}
```

Sin embargo, si el contenido del bucle fuera más complejo quizás sería menos legible. Otra forma alternativa sería la siguiente, donde evitamos el uso de comprobaciones if anidadas dentro del bucle y dividimos en una primera parte del bucle y una segunda parte:

```
for (let i = 0; i < 5; i++) {
  console.log("Valor de i:", i);
}

for (let i = 6; i < 11; i++) {
  console.log("Valor de i:", i);
}
```

Cuidado con los bucles infinitos

Como ves, un mismo problema se puede solucionar casi siempre de múltiples formas, cada una con sus ventajas y desventajas. Ten mucho cuidado si haces este ejemplo con continue y con un bucle while, puesto que puedes crear un bucle infinito:

```
let i = 0;

while (i < 11) {

  if (i == 5) {
    continue;       // ✕ CUIDADO, bucle infinito
  }

  console.log("Iteración número ", i);
  i = i + 1;
}

console.log("Bucle finalizado.")
```

En este ejemplo, observa que el incremento que tenemos en la última línea dentro del bucle i = i + 1;, a diferencia del bucle for, no ocurre en el bucle while dentro del if, por lo que en la iteración 5, se entrará en el bucle if, el continue hará que saltes al principio del bucle pero no se incrementará el contador porque en ningún caso estamos pasando por el incremento.

Por esa razón, nos quedaremos en un eterno bucle infinito, salvo que incluyamos un incremento dentro del if y antes del continue.

La diferencia del continue en el bucle while y en el bucle for, es que en el primero, el incremento se hace de forma manual y se tiene que llegar explícitamente a esa parte para incrementar el valor del contador. En el caso del for, el incremento va incluido en la parte entre paréntesis, por lo que al hacer el continue se incrementa automáticamente.

Interrumpir el bucle

De la misma forma que tenemos un continue que interrumpe la iteración y vuelve al inicio a evaluar la condición, tenemos un break que nos permite interrumpir el bucle y abandonarlo. Esto puede ser bastante útil cuando queremos que se abandone el bucle por una condición especial.

Por ejemplo, cambiemos el continue del ejemplo anterior por el break:

```
for (let i = 0; i < 11; i++) {
  if (i == 5) {
    break;
  }

  console.log("Valor de i:", i);
}
```

En este caso, se van a realizar las iteraciones desde el 0 al 4, y cuando lleguemos a la iteración 5, se entrará en el if y como cumple su condición, se hará un break y se abandonará el for, continuando el resto del programa.

```
let i = 0;

while (i < 11) {
  if (i == 5) {
    break;
  }

  console.log("Iteración número ", i);
  i = i + 1;
}

console.log("Bucle finalizado.")
```

Observa que, en este caso, no hay conflicto con los bucles while, ya que el break no vuelve a evaluar la condición del bucle, sino que directamente abandona el bucle while, por lo que es seguro utilizarlo tanto en for como en while.

No obstante, recuerda que estas interrupciones sólo deben usarse en casos muy específicos. Siempre es preferible que el bucle sea predecible y no tenga demasiadas situaciones excepcionales que interrumpan el flujo, ya que a la larga son difíciles de modificar y mantener.

Variables y constantes

En javascript es muy sencillo declarar y utilizar variables, pero aunque sea un procedimiento simple, hay que tener una serie de conceptos previos muy claros antes de continuar para evitar futuras confusiones, sobre todo si estamos acostumbrados a otros lenguajes más tradicionales.

Variables

En programación, las variables son espacios o «compartimentos» donde se puede guardar información y asociarla a un determinado nombre al que haremos referencia durante el programa.

De esta forma, cada vez que se consulte ese nombre posteriormente, te devolverá la información que contiene. La primera vez que se realiza este paso se suele llamar inicializar una variable.

Inicialización

En Javascript, utilizaremos la palabra clave let para inicializar variables. Si una variable no se inicializa con un valor concreto, contendrá un valor especial llamado undefined, que significa que su valor no está definido aún, o lo que es lo mismo, que no contiene información:

```
let a;          // Declaramos una variable con nombre "a", pero no le asociamos contenido.
let b = 0;      // Declaramos una variable con nombre "b", y le asociamos el número 0.

console.log(b); // Muestra 0 (el valor guardado en la variable "b")
console.log(a); // Muestra "undefined" (no hay valor guardado en la variable "a")
```

Como se puede observar, hemos utilizado console.log() para consultar la información que contienen las variables indicadas.

OJO: Las mayúsculas y minúsculas en los nombres de las variables de Javascript importan. **No es lo mismo una variable llamada precio que una variable llamada Precio,** pueden contener valores diferentes.

Si tenemos que declarar muchas variables consecutivas, una buena práctica suele ser escribir sólo el primer let y separar por comas las diferentes variables con sus respectivos contenidos (método 3). Aunque se podría escribir todo en una misma línea (método 2), con el último método el código es mucho más fácil de leer:

```
// Método 1: Declaración de variables de forma independiente
let a = 3;
let c = 1;
let d = 2;

// Método 2: Declaración masiva de variables en el mismo let
let a = 3, c = 1, d = 2;

// Método 3: Igual al anterior, pero mejorando la legibilidad del código
let a = 3,
    c = 1,
    d = 2;
```

Antiguamente, en Javascript se utilizaba la palabra clave var para inicializar variables. Aunque te lo puedes encontrar o pueden existir situaciones donde puede ser útil, hoy en día no tiene demasiado sentido utilizar var en nuestro código.

Reasignación

Como su propio nombre indica, una variable puede variar su contenido, ya que, aunque contenga una cierta información, se puede volver a cambiar.

A esta acción ya no se le llamaría «inicializar» una variable, sino «redeclarar» o «reasignar» una variable, ya que le estamos cambiando su contenido.

En el código se puede diferenciar porque se omite el let (la variable ya existe):

```
// Inicializamos la variable "a" al valor 40.
let a = 40;

// Ahora, hemos redeclarado la variable a, que pasa a contener 50 en lugar de 40.
a = 50;
```

Constantes

En ECMAScript 2015 **se añade la palabra clave const, que permite crear «compartimentos» similares a una variable, salvo que los valores que contiene no pueden ser modificados**. Esto tiene varios matices, así que vamos a explicarlo poco a poco. Observa el siguiente ejemplo:

```
const name = "Manz";
console.log(name);

name = "Paco";  // Uncaught TypeError: Assignment to constant variable.
```

Como puedes ver, la diferencia respecto al let es que const asigna un valor y no permite reasignarlo o redeclararlo posteriormente. Además, tampoco puedes crear una constante sin indicarle un valor concreto:

```
const surname;  // Uncaught SyntaxError: Missing initializer in const declaration
```

En Javascript, es muy habitual declarar las variables con const para anunciar implícitamente que esos valores no cambiarán a lo largo del programa, **y en el caso que tengamos intenciones de que cambien, declararlas con let. En esos casos, se suele utilizar camelCase como estrategia de nombrado**.

Existen ciertos casos en los que se suele utilizar SNAKE_UPPER_CASE, y es en la situación en las que tenemos constantes con valores que no van a cambiar durante el programa, pero que el

programador estará cambiando continuamente durante el desarrollo del programa para ajustar cambios. Estas constantes se suelen declarar al principio del programa, para que sea más sencillo encontrarlas y tenerlas en cuenta.

```
const MAX_ITERATIONS = 10;

for (let i = 0; i < MAX_ITERATIONS; i++) {
  console.log("Iteración número ", i);
}

console.log("Bucle finalizado.");
```

Observa que, en este caso, si tuviéramos que cambiar el número de iteraciones del bucle, en lugar de tener que modificar el código del programa, sólo tendríamos que modificar la constante inicial.

No hay que confundir constantes con **valores inmutables**, ya que como veremos más adelante, **algunas constantes (objetos, arrays...) sí que pueden ser alteradas, aunque nunca reasignadas**

Tipos de datos

En Javascript, al igual que en la mayoría de los lenguajes de programación, **al declarar una variable y guardar su contenido, también le estamos asignando un tipo de dato**.

En algunos lenguajes se hace de forma explícita, y en algunos otros lenguajes se hace de forma implícita.

El tipo de dato no es más que la naturaleza de su contenido: contenido numérico, contenido de texto, etc...

¿Qué tipos de lenguajes existen?

A grandes rasgos, nos podemos encontrar con dos tipos de lenguajes de programación:

Lenguajes estáticos: Cuando creamos una variable, debemos indicar el tipo de dato del valor que va a contener.

En consecuencia, el valor asignado finalmente, siempre deberá ser del tipo de dato que hemos indicado (si definimos que es un número debe ser un número, si definimos que es un texto debe ser un texto, etc...).

Lenguajes dinámicos: Cuando creamos una variable, no es necesario indicarle el tipo de dato que va a contener.

El lenguaje de programación se encargará de deducir o inferir el tipo de dato (dependiendo del valor que le hayamos asignado).

Javascript pertenece a los lenguajes dinámicos, ya que automáticamente detecta de qué tipo de dato se trata en cada caso, dependiendo del contenido que le hemos asignado a la variable.

En los lenguajes dinámicos, realmente el tipo de dato se asocia al valor (y no a la variable).

Así es mucho más fácil entender que a lo largo del programa, dicha variable puede «cambiar» a tipos de datos diferentes, ya que la restricción del tipo de dato está asociada al valor y no a la variable en sí.

No obstante, para simplificar, solemos hablar de variables y sus respectivos tipos de datos.

¿Qué es mejor?

Para algunos desarrolladores (nóveles, que necesitan flexibilidad, etc...) los lenguajes donde no tienes que indicar el tipo de dato les resulta mucho más cómodo y agradable, ya que es mucho más rápido y productivo declarar variables sin tener que preocuparte del tipo de dato que necesitan.

Sin embargo, para otros desarrolladores (tradicionales, backend, etc...) los lenguajes donde no tienes que indicar el tipo de dato les resulta una desventaja, ya que pierden el control de la información almacenada y en muchas ocasiones esto puede desembocar en problemas o situaciones inesperadas.

Javascript

Es importante recalcar que en Javascript, aunque no tengamos que indicar el tipo de dato de cada variable, los tipos de datos existen, se puede acceder a ellos e incluso convertir entre un tipo de dato y otro.

```
const valueAsText = "50";
const valueAsNumber = 50;
const convertedValue = Number("50");

console.log(valueAsText);    // "50"
console.log(valueAsNumber);  // 50
console.log(convertedValue); // 50
```

Observa que en el ejemplo anterior, la constante valueAsText tiene un tipo de dato de texto, mientras que la constante valueAsNumber tiene un tipo de dato numérico. Hemos utilizado Number() para forzar un texto y convertirlo a otro tipo de dato (numérico).

Typescript

Aunque en Javascript existen mecanismos para convertir o forzar los tipos de datos de las variables, muchos programadores prefieren indicar explícitamente los tipos de datos, ya que esto les aporta cierta confianza y seguridad, especialmente en aplicaciones muy grandes o donde hay muchos programadores diferentes modificando código.

Estos desarrolladores suelen optar por utilizar lenguajes como **Typescript**, que no es más que «varias capas de características añadidas» sobre Javascript, convirtiéndolo en un lenguaje más parecido a, por ejemplo, Java:

```
const valueAsText : string = "50";
const valueAsNumber : number = 50;
```

Sin embargo, es importante recordar que Typescript no es un lenguaje que el navegador entiende, **sino un lenguaje que añade funcionalidades que el programador necesita o prefiere, y luego es convertido a Javascript para que el navegador pueda entenderlo.**

En muchas ocasiones (y de manera informal) también se suele hacer referencia a lenguajes tipados (tipado fuerte, o fuertemente tipado) o lenguajes no tipados (tipado débil, debilmente tipado), para indicar si el lenguaje requiere indicar manualmente el tipo de dato de las variables o no, respectivamente.

Ámbitos o contextos

Un tema complejo en el mundo de la programación es el de los ámbitos, scopes o contextos. Cuando comenzamos a aprender sobre variables y constantes, es bueno aprender y entender bien los ámbitos, ya que es la zona de alcance que tiene una variable.

En principio, existen dos ámbitos muy bien definidos:

- **Ámbito global**: Existe a lo largo de todo el programa o aplicación.
- **Ámbito local**: Existe sólo en una pequeña región del programa.

Normalmente, esto se determina muy fácilmente observando las llaves { y } (curly braces), que abren un nuevo ámbito o contexto.

Ámbitos de variables

Cuando inicializamos una variable al principio de nuestro programa y le asignamos un valor, ese valor generalmente está disponible a lo largo de todo el programa.

Sin embargo, esto puede variar dependiendo de múltiples factores. Se conoce como ámbito de una variable a la zona donde esa variable existe.

Por ejemplo, si consultamos el valor de una variable antes de inicializarla, no existe:

```
console.log(e);  // Uncaught ReferenceError: e is not defined

let e = 40;
console.log(e);  // 40 (existe porque ya se ha inicializado en la línea anterior)
```

En el ejemplo anterior, el ámbito de la variable `e` **comienza a partir de su inicialización y "vive" hasta el final del programa. A esto se le llama ámbito global** y es el caso más sencillo. Observa que en la primera línea la variable `e` no existe y da un error**, pero tras la linea del let**, si devuelve su valor porque ahí si existe.

Sin embargo, esto se puede ir complicando a medida que vamos abriendo llaves de instrucciones como if, else, for o while. Observa el siguiente ejemplo:

```
let a = 1;
console.log(e);  // Uncaught ReferenceError: e is not defined

if (a == 1) {
  let e = 40;
  console.log(e);   // 40, existe
}

console.log(e);  // Uncaught ReferenceError: e is not defined
```

Observa que la variable **a** existe en un ámbito global, mientras que la variable **e** sólo existe en el interior del **if**: **es un ámbito local.**

Scope o ámbitos: Simba meme

Es puede volverse un poco más complicado dependiendo de si **utilizar let, const o var, o dependiendo de la anidación.** Lo iremos viendo a medida que avancemos.

Ámbitos de variables: let y const

En las versiones modernas de Javascript (a partir de ECMAScript 2015), se introduce la palabra clave **let** en sustitución de la antigua **var** para declarar **variables y const** para declarar constantes.

Tanto con let como con const, estaremos utilizando los ámbitos clásicos de programación: ámbito global y ámbito local.

Veamos otro ejemplo, esta vez utilizando un bucle for:

```
console.log("Antes: ", p);          // En este punto, p no está definida
for (let p = 0; p < 3; p++) {
  console.log("Valor de p: ", p);   // Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p);        // En este punto, vuelve a no estar definida
```

Varios detalles:

- Utilizando **let en el bucle for**, la variable **p** sólo está definida dentro del bucle (**ámbito local**)
- Observa que tanto antes como después del bucle, p no existe.

Aunque omitamos las llaves del for (en este caso es posible porque sólo contiene una línea), los ámbitos siguen existiendo. Hacemos referencia a las llaves porque se ve mejor, pero lo que importa es si se declara la variable dentro del bucle for o no.

Ámbitos de variables: var (legacy)

Aunque declarar variables con **var** ya se considera legacy (**obsoleto**) y no debe usarse, si te interesa saber un poco más por si te lo encuentras, en este enfoque tradicional de Javascript, existen ámbitos diferentes: **el ámbito global y el ámbito a nivel de función**.

La diferencia se puede ver claramente en el **uso de un bucle for con var** (a diferencia del que vimos anteriormente con let):

```
console.log("Antes: ", p);          // En este punto, p vale undefined
for (var p = 0; p < 3; p++) {
  console.log("Valor de p: ", p);   // Aquí, p estará definida como 0, 1, 2
}
console.log("Después: ", p);        // Después: 3 (WTF!)
```

Esto es algo que a un programador acostumbrado a otros lenguajes con ámbitos tradicionales le puede explotar la cabeza. Si utilizamos var la variable p sigue existiendo fuera del bucle, aunque haya sido declarada en su interior. Esto ocurre porque en ese caso se usa un ámbito a nivel de función.

Veamos este otro ejemplo (necesitaras conocer el concepto de función, por lo que si no lo conoces, quizás necesites volver más tarde cuando hayas visto ese tema):

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1

function x() {
  console.log(a); // En esta línea el valor de "a" es undefined
  var a = 5; // Aquí creamos una variable "a" a nivel de función

  console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 1 (ámbito global)
}

x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 1
```

En el ejemplo anterior vemos que **el valor de a** dentro de una función no es el 1 inicial, sino que estamos en otro ámbito diferente donde la variable a anterior no existe: un ámbito a nivel de función.

Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el ámbito de la propia función.

Estamos usando el objeto especial window para acceder directamente al ámbito global independientemente de donde nos encontremos. **Esto ocurre así porque las variables globales se almacenan**

dentro del objeto window (la pestaña actual del navegador web). Hoy en día, en lugar de utilizar window sería preferible utilizar globalThis.

Sin embargo, si eliminamos el var de la línea var a = 5 del ejemplo anterior, observa la diferencia:

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1

function x() {
  console.log(a); // En esta línea el valor de "a" es 1
  a = 5; // Aquí creamos una variable "a" en el ámbito anterior

  console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 5 (ámbito global)
}

x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 5
```

En este ejemplo se omite el var dentro de la función, y vemos que en lugar de crear una variable en el ámbito de la función, se modifica el valor de la variable a a nivel global. Dependiendo de donde y como accedamos a la variable a, obtendremos un valor u otro.

Siempre que sea posible se debería utilizar let y const, en lugar de var. Declarar variables mediante var se recomienda en fases de aprendizaje o en el caso de que se quiera mantener compatibilidad con navegadores muy antiguos utilizando ECMAScript 5, sin embargo, hay estrategias mejores a seguir que utilizar var en la actualidad.

¿Qué es una función?

En programación, cuando nuestro código se va haciendo cada vez más grande, necesitaremos buscar una forma de organizarlo y prepararnos para reutilizarlo y no repetir innecesariamente las mismas tareas.

Para ello, un primer recurso muy útil son las funciones.

¿Qué es una función?

Las funciones nos permiten agrupar líneas de código en tareas con un nombre, para que, posteriormente, podamos hacer referencia a ese nombre para realizar todo lo que se agrupe en dicha tarea. Para usar funciones hay que hacer 2 cosas:

- **Declarar la función:** **Preparar la función, darle un nombre y decirle las tareas que realizará.**
- **Ejecutar la función:** **«Llamar» a la función para que realice las tareas de su contenido.**
-

Declaración

En el siguiente ejemplo veremos la declaración de una función llamada saludar:

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  console.log("Hola, soy una función");
}
```

El contenido de la función es una línea que mostrará por consola un saludo. Sin embargo, si escribimos estas 4-5 líneas de código en nuestro programa, no mostrará nada por pantalla.

Esto ocurre así porque solo hemos declarado la función (le hemos dicho que existe), **pero aún nos falta el segundo paso, ejecutarla**, que es realmente cuando se realizan las tareas de su contenido.

Ejecución

Veamos, ahora sí, el ejemplo completo con declaración y ejecución:

```
// Declaración de la función "saludar"
function saludar() {
  // Contenido de la función
  console.log("Hola, soy una función");
}

// Ejecución de la función
saludar();
```

En este ejemplo hemos declarado la función y además, hemos ejecutado la función (en la última línea) llamándola por su nombre y seguida de ambos paréntesis, que nos indican que es una función. En este ejemplo, si se nos mostraría en la consola Javascript el mensaje de saludo.

Ejemplo

Veamos un primer ejemplo que muestre en la consola Javascript la tabla de multiplicar del 1:

```
// Tabla de multiplicar del 1
console.log("1 x 0 = ", 1 * 0);
console.log("1 x 1 = ", 1 * 1);
console.log("1 x 2 = ", 1 * 2);
console.log("1 x 3 = ", 1 * 3);
console.log("1 x 4 = ", 1 * 4);
console.log("1 x 5 = ", 1 * 5);
console.log("1 x 6 = ", 1 * 6);
console.log("1 x 7 = ", 1 * 7);
console.log("1 x 8 = ", 1 * 8);
console.log("1 x 9 = ", 1 * 9);
console.log("1 x 10 = ", 1 * 10);
```

Este primer ejemplo funciona perfectamente, sin embargo, no estamos aprovechando las ventajas de la programación, sino que hemos hecho todo el trabajo escribiendo 10 líneas de código.

Vamos a utilizar un bucle for para ahorrarnos tantas líneas de código:

```
// Declaración
for (let i = 0; i < 11; i++) {
  console.log("1 x", i, "=", 1 * i);
}
```

Esto está mucho mejor. Hemos resumido 11 líneas de código en prácticamente 2 líneas utilizando un bucle for que va de 0 a 10 incrementando de 1 en 1. Incluso, recuerda que cuando las llaves sólo contienen una línea, se pueden omitir.

Otro detalle importante a mencionar es que, en muchos casos, a los programadores les resulta más intuitivo establecer la condición del bucle a i <= 10 que establecerla a i < 11. Son exactamente equivalentes, pero en el caso actual, la primera parece más «lógica», ya que vemos visualmente el número de la última iteración.

Pero aún no hemos utilizado funciones, así que vamos a modificar nuestro ejemplo para usar una. Imaginemos que ahora nuestro objetivo es mostrar la tabla de multiplicar del 1 varias veces (3 veces para ser exactos).

La primera aproximación para hacer eso que se nos ocurriría sería hacer lo siguiente:

```
// Primera vez
for (let i = 0; i ⩽ 10; i++) console.log("1 x", i, "=", 1 * i);

// Segunda vez
for (let i = 0; i ⩽ 10; i++) console.log("1 x", i, "=", 1 * i);

// Tercera vez
for (let i = 0; i ⩽ 10; i++) console.log("1 x", i, "=", 1 * i);
```

Pero volvemos a tener el mismo problema del primer ejemplo. Estamos repitiendo el mismo código varias veces, complicándolo y volviéndolo más «feo».

Además, si tuviéramos que hacer modificaciones en uno de los bucles habría también que repetir el trabajo 2 veces más, por cada uno de los otros bucles.

Veamos ahora como obtener el mismo resultado, pero utilizando bucles y funciones, sin repetir varias veces las mismas tareas:

```
// Declaración de la función tablaDelUno()
function tablaDelUno() {
  for (let i = 0; i ⩽ 10; i++) console.log("1 x", i, "=", 1 * i);
}

// Bucle que ejecuta 3 veces la función tablaDelUno()
for (intento = 0; intento < 3; intento++) tablaDelUno();
```

En este ejemplo se declara la función, que mostrará la tabla de multiplicar del uno. Posteriormente, realizamos un bucle con la variable intento de 0 a 2 (3 repeticiones) para llamar la función 3 veces, y mostrar así la tabla de multiplicar cada vez.

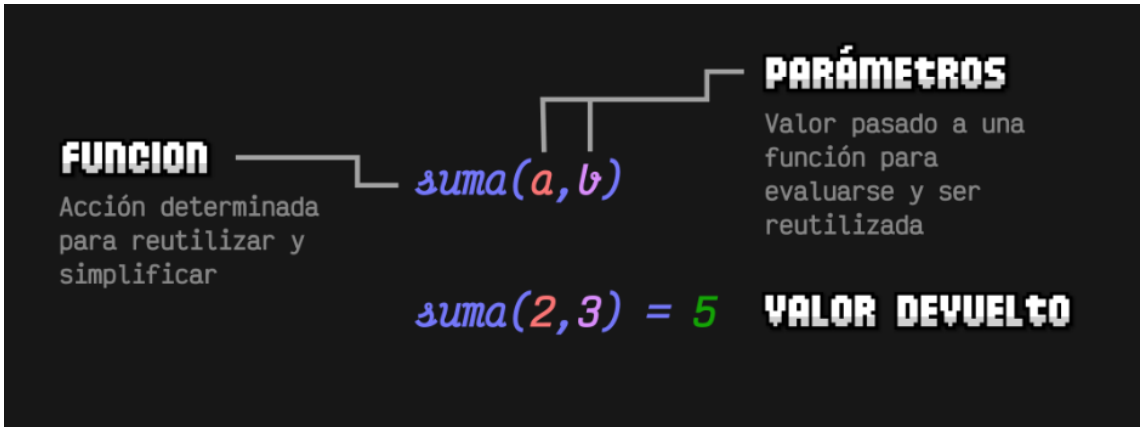
Pero... ¿No sería más interesante mostrar las 3 primeras tablas de multiplicar (Tabla del uno, del dos y del tres)? Para hacer eso, primero tenemos que conocer los parámetros en funciones, que veremos en el siguiente artículo.

Parámetros de una función

En el apartado anterior, hemos visto la parte más simple de función, que es abstraer información para simplificar y reducir nuestro código. Sin embargo, una de las virtudes más interesantes de una función es la de poder reutilizarla para múltiples tareas. Y para ello, tenemos que saber que son los parámetros de entrada y de salida de una función.

¿Qué son los parámetros?

Las funciones son mucho más flexibles y potentes de lo que hemos visto hasta ahora. A las funciones se les pueden pasar parámetros, que no son más que variables que les pasamos desde fuera hacia dentro de la función. Además, también podemos hacer que la función realice sus tareas y nos devuelva un resultado hacia el exterior de la función.



Funciones y parámetros de una función

Veamos el siguiente ejemplo, donde utilizamos el parámetro hasta para indicar hasta donde debe llegar:

```
// Declaración
function tablaDelUno(hasta) {
  for (let i = 0; i <= hasta; i++) {
    console.log("1 x", i, "=", 1 * i);
  }
}

// Ejecución
tablaDelUno(10); // Tabla del 1
tablaDelUno(5);  // Tabla del 1
```

Como podemos ver, en el interior de los paréntesis de la función se ha indicado una variable llamada hasta. Esa variable contiene el valor que se le da a la hora de ejecutar la función, que, en este ejemplo, si nos fijamos bien, se ejecuta dos veces: una con valor 10 y otra con valor 5.

Analicemos este código:

- Ejecutamos la función tablaDelUno(10).
- En la función tablaDelUno, el parámetro hasta valdrá 10.
- Por lo tanto, haremos un bucle for desde 0 hasta 10, incrementando de 1 en 1.
- Se mostrará la tabla de multiplicar del uno, desde el 0 hasta el 10.

Al terminar ejecutaremos la segunda función:

- Ejecutamos la función tablaDelUno(5).
- En la función tablaDelUno, el parámetro hasta valdrá 5.
- Por lo tanto, haremos un bucle for desde 0 hasta 5, incrementando de 1 en 1.
- Se mostrará la tabla de multiplicar del uno, desde el 0 hasta el 5.

La idea de las funciones es enfocarnos en el código de la declaración, y una vez lo tengamos funcionando, nos podemos olvidar de él porque está encapsulado dentro de la función. Simplemente tendremos que recordar el nombre de la función y los parámetros que hay que pasarle. Esto hace que sea mucho más fácil trabajar con el código.

Parámetros múltiples

Hasta ahora sólo hemos creado una función con 1 parámetro, pero una función de Javascript puede tener muchos más parámetros. Vamos a crear otro ejemplo, mucho más útil donde convertimos nuestra función en algo más práctico y útil:

```
// Declaración
function tablaMultiplicar(tabla, hasta) {
  for (let i = 0; i <= hasta; i++) {
    console.log(tabla, "x", i, "=", tabla * i);
  }
}

// Ejecución
tablaMultiplicar(1, 10); // Tabla del 1
tablaMultiplicar(5, 10); // Tabla del 5
```

En este ejemplo, hemos modificado nuestra función tablaDelUno() por esta nueva versión que hemos cambiado de nombre a tablaMultiplicar().

Esta función necesita que le pasemos dos parámetros: tabla (la tabla de multiplicar en cuestión) y hasta (el número hasta donde llegará la tabla de multiplicar).

De esta forma, las dos llamadas para ejecutar la función mostrarán por la consola la tabla de multiplicar del 1 y del 5.

Podemos añadir más parámetros a la función según nuestras necesidades. Es importante recordar que el orden de los parámetros es importante y que los nombres de cada parámetro no se pueden repetir en una misma función.

Parámetros por defecto

Es posible que en algunos casos queramos que ciertos parámetros tengan un valor sin necesidad de escribirlos en la ejecución. Es lo que se llama un valor por defecto.

En nuestro ejemplo anterior, nos podría interesar que la tabla de multiplicar llegue siempre hasta el 10, ya que es el comportamiento por defecto.

Si queremos que llegue hasta otro número, lo indicamos explícitamente, pero si lo omitimos, queremos que llegue hasta 10. Esto se haría de la siguiente forma:

```
function tablaMultiplicar(tabla, hasta = 10) {
  for (let i = 0; i ⩽ hasta; i++) {
    console.log(tabla, "x", i, "=", tabla * i);
  }
}

// Ejecución
tablaMultiplicar(2);      // Tabla del 2, del 0 al número 10
tablaMultiplicar(2, 15);  // Tabla del 2, del 0 al número 15
```

De esta forma nos ahorramos tener que escribir los valores en la ejecución de la función, si en la mayoría de los casos va a tomar ese valor.

Devolución de valores

Hasta ahora hemos utilizado funciones simples que realizan acciones o tareas (en nuestro caso, mostrar por consola), pero habitualmente, lo que buscamos es que esa función realice una tarea y nos devuelva la información al exterior de la función, para así utilizarla o guardarla en una variable, que utilizaremos posteriormente para nuestros objetivos.

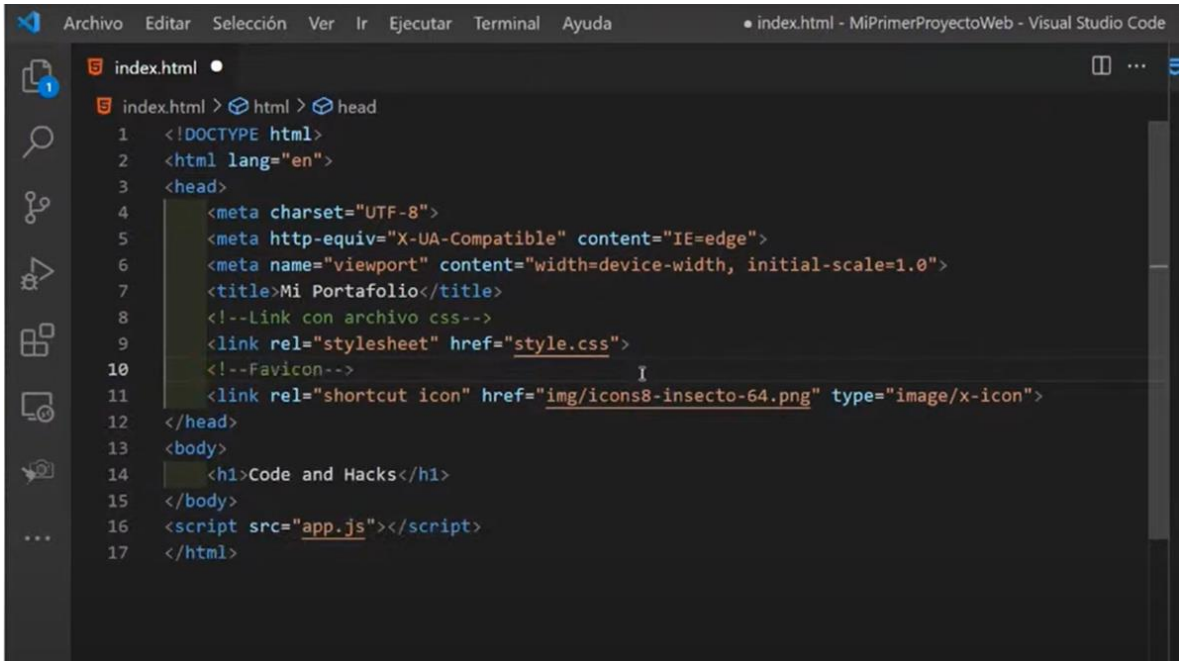
Para ello, se utiliza la palabra clave return, **que suele colocarse al final de la función, ya que con dicha devolución terminamos la ejecución de la función** (si existe código después, nunca será ejecutado).

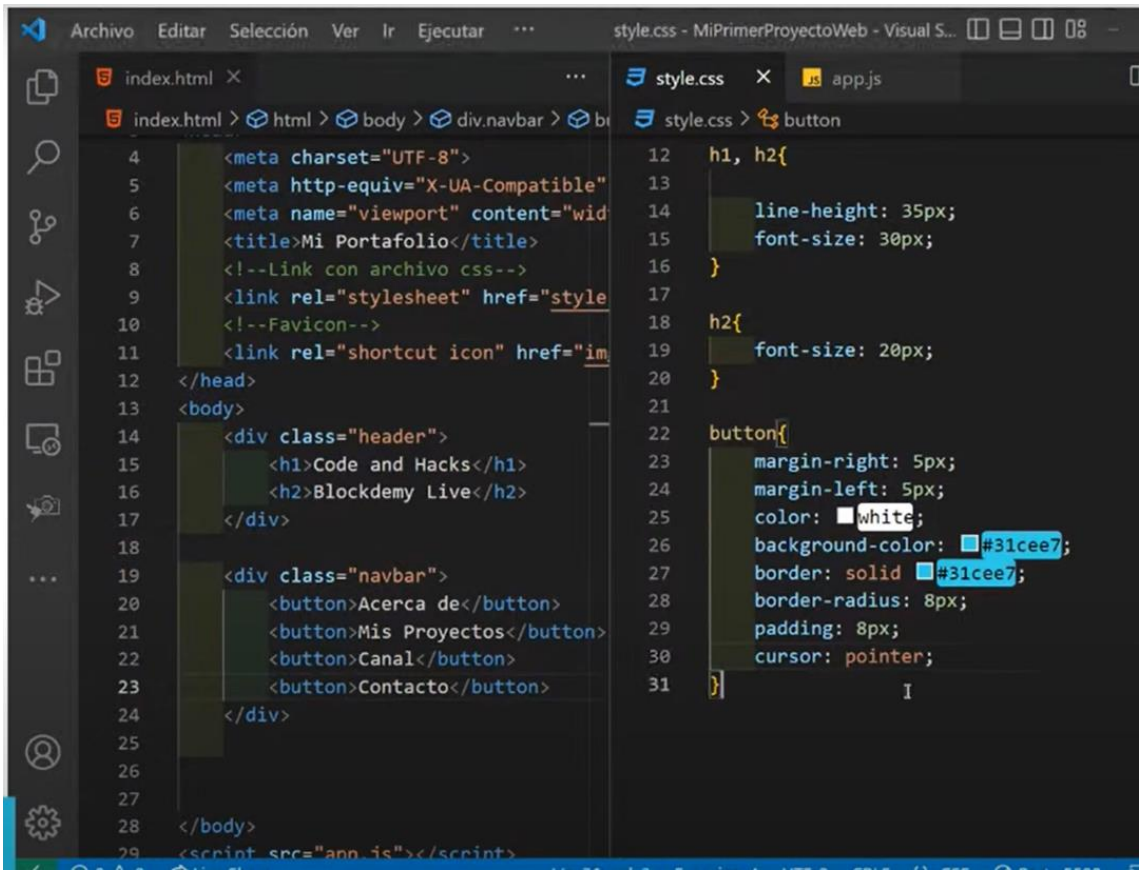
Veamos un ejemplo con una operación muy sencilla, para verlo claramente:

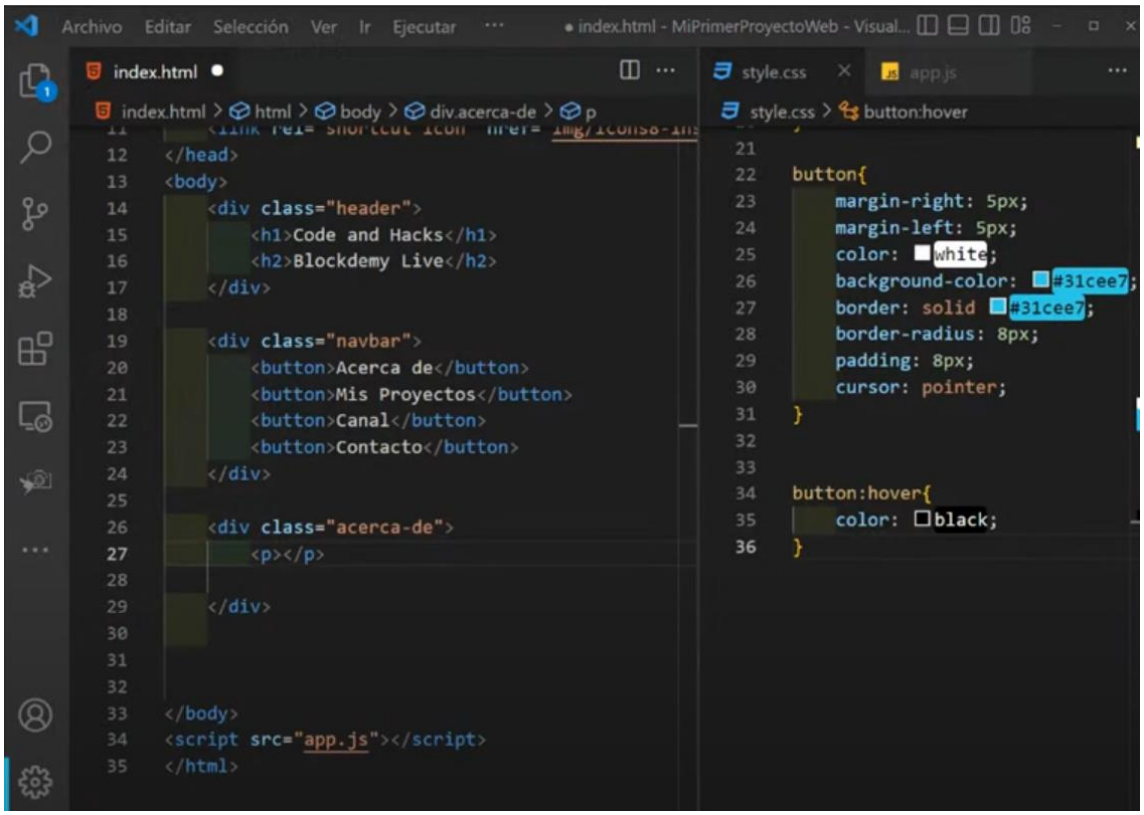
```
function sumar(a, b) {
  return a + b;                  // Devolvemos la suma de a y b al exterior
  console.log("Suma realizada."); // Este código nunca se ejecutará
}

// Ejecución
const resultado = sumar(5, 5);           // Se guarda 10 en la variable resultado
console.log("Resultado = ", resultado);  // Mostramos el resultado por consola
```

Como podemos ver, esto nos permite crear funciones más modulares y reutilizables que podremos utilizar en multitud de casos, ya que la información se puede enviar al exterior de la función y utilizarla junto a otras funciones o para otros objetivos.

Ejercicio de Ejemplo



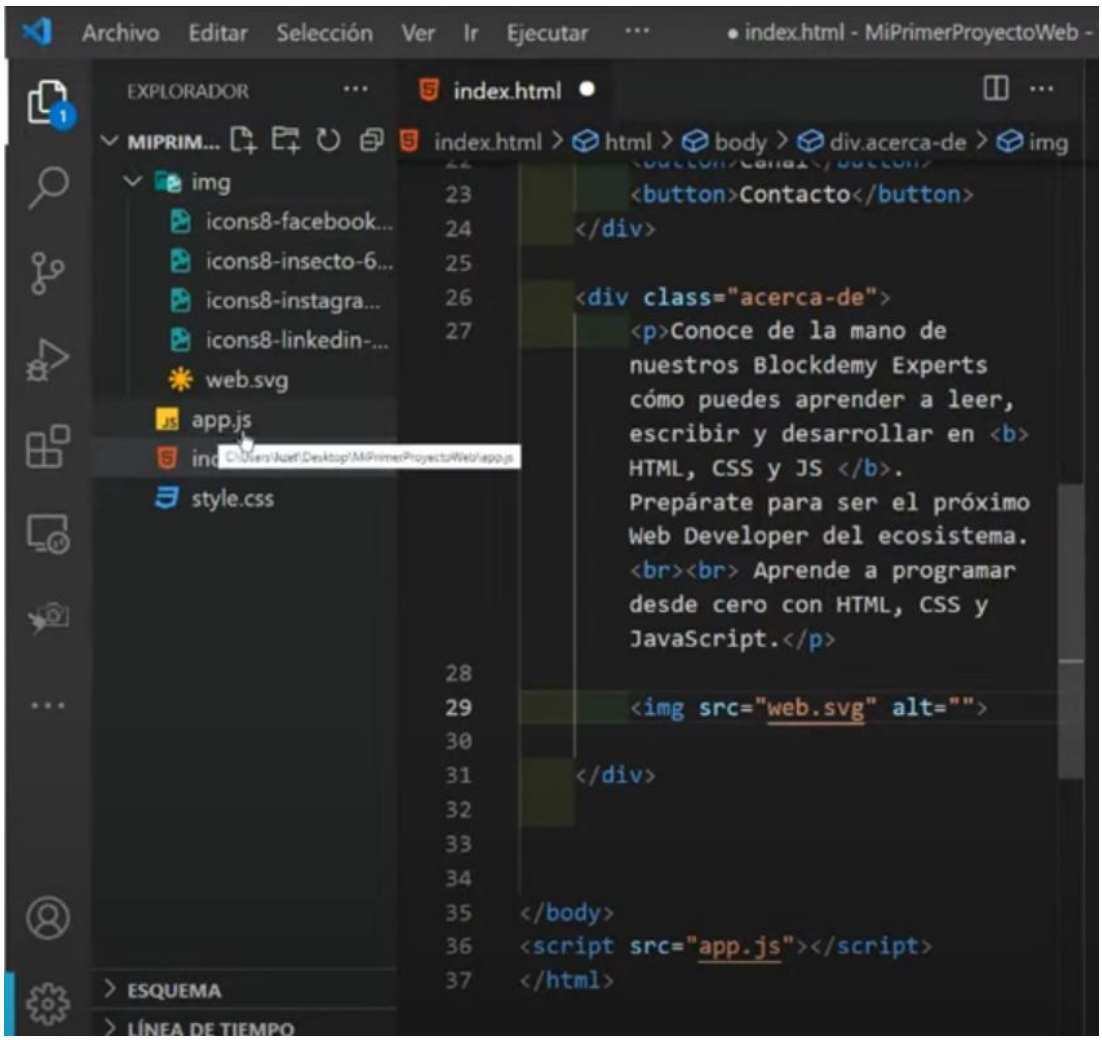



Archivo Editar Selección Ver Ir Ejecutar
index.html - MiPrimerProyectoWeb - Visual...
index.html
style.css
app.js
index.html > html > body > div.acerca-de > p
style.css > button:hover
</head>
<body>

<h1>Code and Hacks</h1>
<h2>Blockdemy Live</h2>

<button>Acerca de</button>
<button>Mis Proyectos</button>
<button>Canal</button>
<button>Contacto</button>

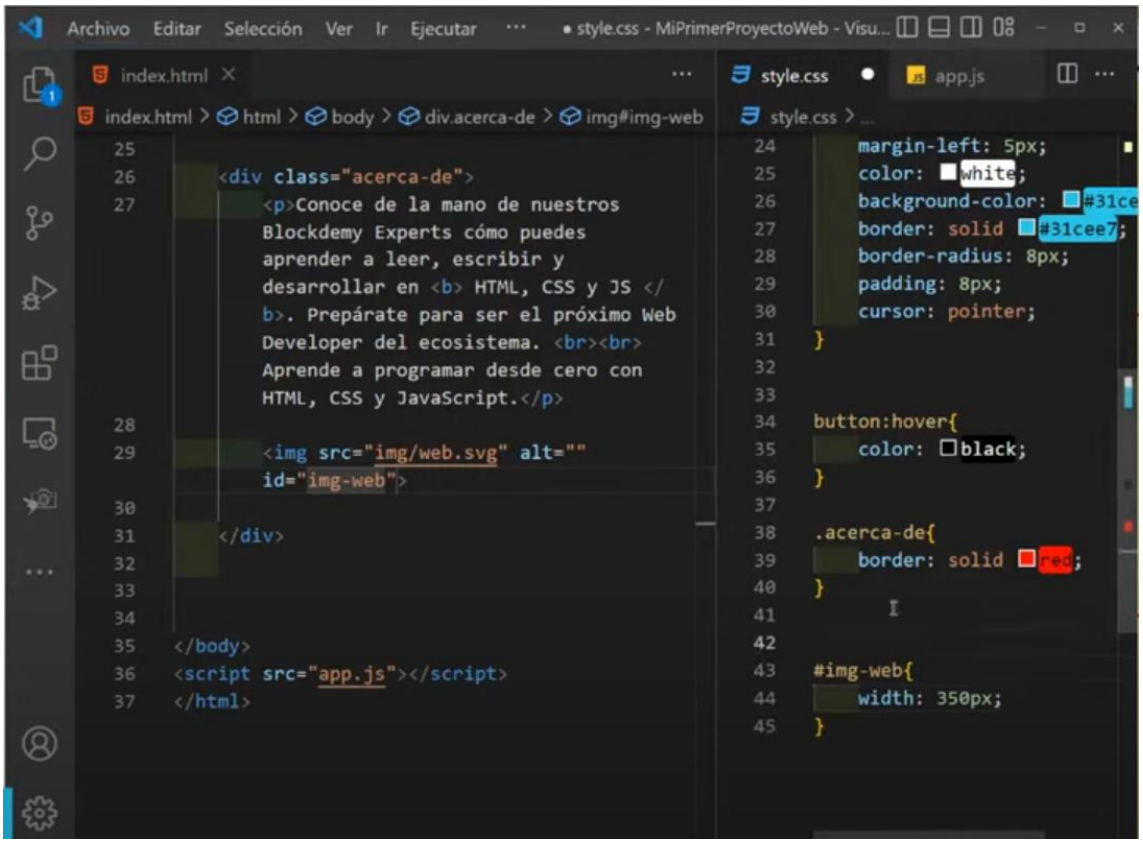
</body>
<script src="app.js"></script>
</html>
button{
margin-right: 5px;
margin-left: 5px;
color: white;
background-color: #31cee7;
border: solid #31cee7;
border-radius: 8px;
padding: 8px;
cursor: pointer;
}
button:hover{
color: black;
}


Archivo Editar Selección Ver Ir Ejecutar
index.html - MiPrimerProyectoWeb -
EXPLORADOR
MIPRIM...
index.html > html > body > div.acerca-de > img
img
icons8-facebook...
icons8-insecto-6...
icons8-instagra...
icons8-linkedin-...
web.svg
app.js
style.css
<button>Contacto</button>

Conoce de la mano de
nuestros Blockdemy Experts
cómo puedes aprender a leer,
escribir y desarrollar en
HTML, CSS y JS .
Prepárate para ser el próximo
Web Developer del ecosistema.

 Aprende a programar
desde cero con HTML, CSS y
JavaScript.
<img src="web.svg" alt="">

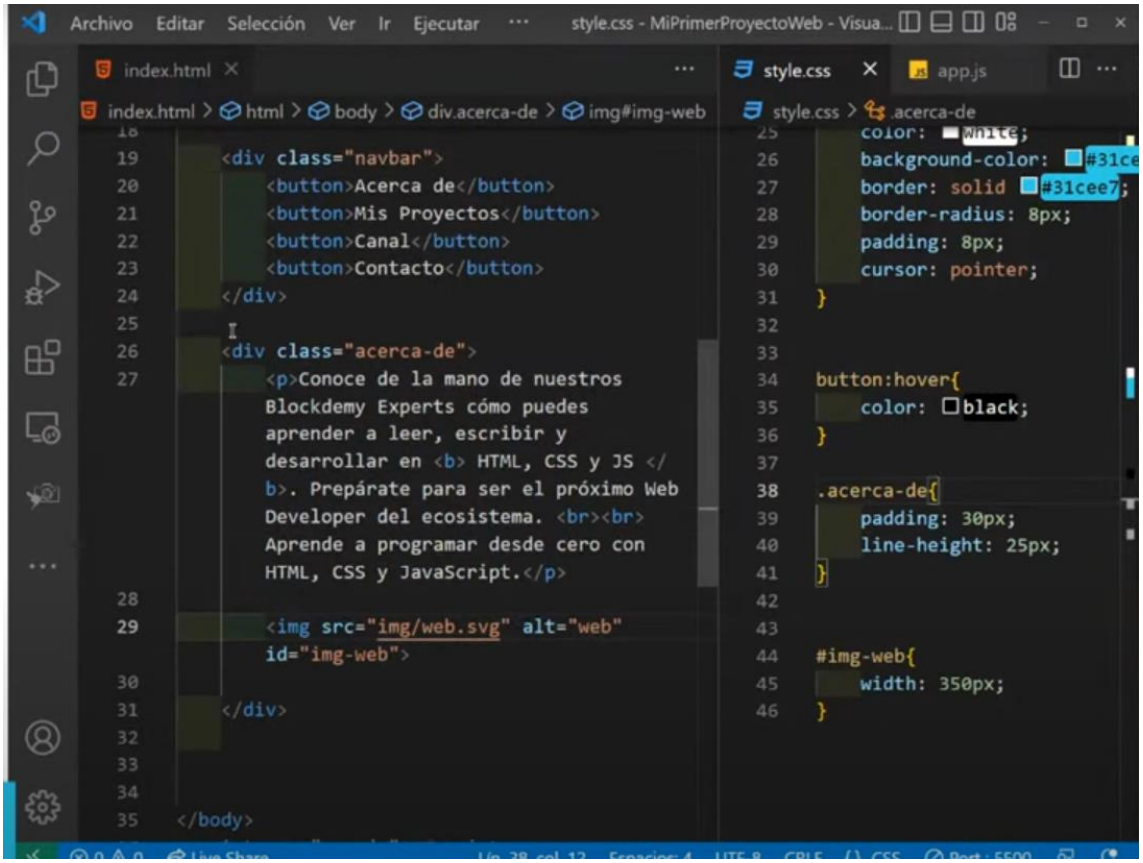
</body>
<script src="app.js"></script>
</html>
ESQUEMA
LÍNEA DE TIEMPO


Archivo Editar Selección Ver Ir Ejecutar ... • style.css - MiPrimerProyectoWeb - Visu...
index.html
index.html > html > body > div.acerca-de > img#img-web

Conoce de la mano de nuestros Blockdemy Experts cómo puedes aprender a leer, escribir y desarrollar en HTML, CSS y JS . Prepárate para ser el próximo Web Developer del ecosistema.

 Aprende a programar desde cero con HTML, CSS y JavaScript.
<img src="img/web.svg" alt="" id="img-web">

</body>
<script src="app.js"></script>
</html>
style.css
app.js
style.css > ...
margin-left: 5px;
color: white;
background-color: #31ce
border: solid #31cee7;
border-radius: 8px;
padding: 8px;
cursor: pointer;
}
button:hover{
color: black;
}
.acerca-de{
border: solid red;
}
#img-web{
width: 350px;
}

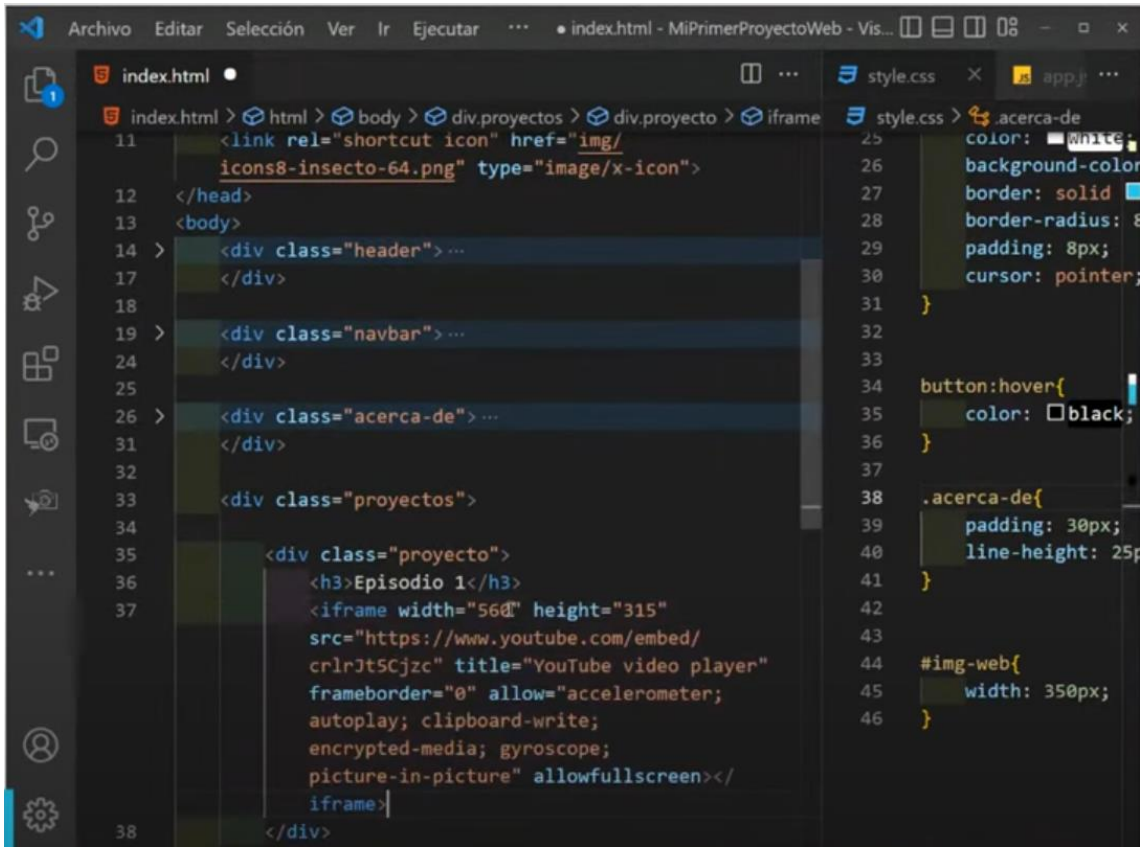

Archivo Editar Selección Ver Ir Ejecutar ... style.css - MiPrimerProyectoWeb - Visua...
index.html
index.html > html > body > div.acerca-de > img#img-web

<button>Acerca de</button>
<button>Mis Proyectos</button>
<button>Canal</button>
<button>Contacto</button>

Conoce de la mano de nuestros Blockdemy Experts cómo puedes aprender a leer, escribir y desarrollar en HTML, CSS y JS . Prepárate para ser el próximo Web Developer del ecosistema.

 Aprende a programar desde cero con HTML, CSS y JavaScript.
<img src="img/web.svg" alt="web" id="img-web">

</body>
style.css
app.js
style.css > .acerca-de
color: white;
background-color: #31ce
border: solid #31cee7;
border-radius: 8px;
padding: 8px;
cursor: pointer;
}
button:hover{
color: black;
}
.acerca-de{
padding: 30px;
line-height: 25px;
}
#img-web{
width: 350px;
}

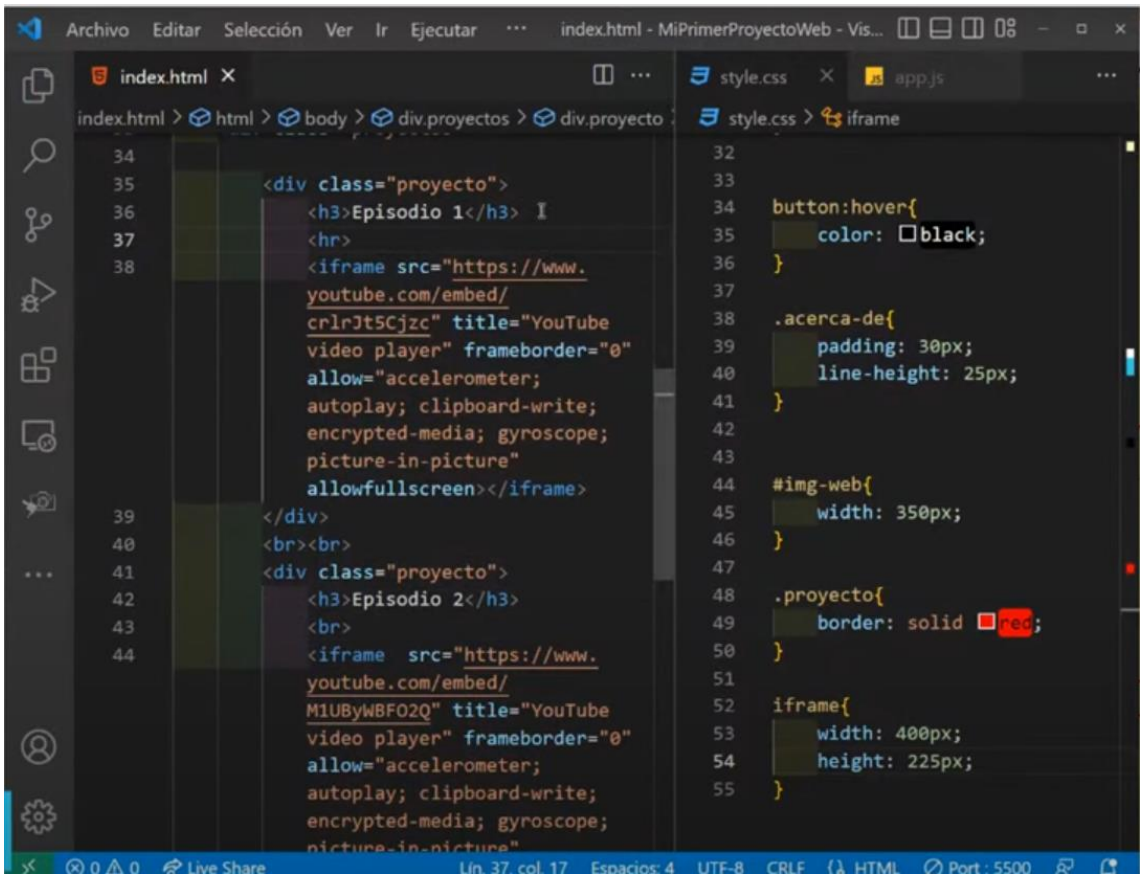

Archivo Editar Selección Ver Ir Ejecutar ... • index.html - MiPrimerProyectoWeb - Vis...
index.html
style.css
app.js
index.html > html > body > div.proyectos > div.proyecto > iframe
style.css > .acerca-de
<link rel="shortcut icon" href="img/icons8-insecto-64.png" type="image/x-icon">
</head>
<body>
...

...

...

<h3>Episodio 1</h3>
<iframe width="560" height="315" src="https://www.youtube.com/embed/crlrJt5Cjzc" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

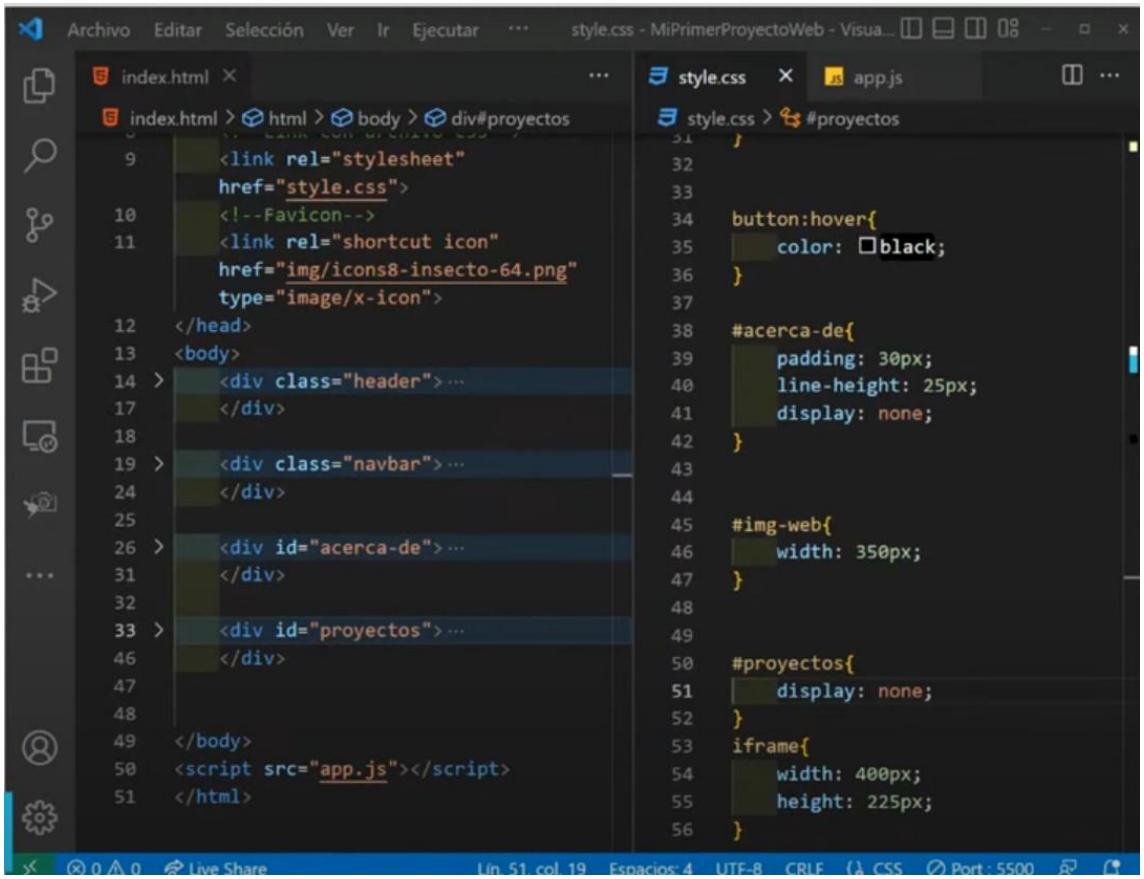
color: white;
background-color
border: solid
border-radius:
padding: 8px;
cursor: pointer;
}
button:hover{
color: black;
}
.acerca-de{
padding: 30px;
line-height: 25p
}
#img-web{
width: 350px;
}


Archivo Editar Selección Ver Ir Ejecutar ... index.html - MiPrimerProyectoWeb - Vis...
index.html
style.css
app.js
index.html > html > body > div.proyectos > div.proyecto
style.css > iframe

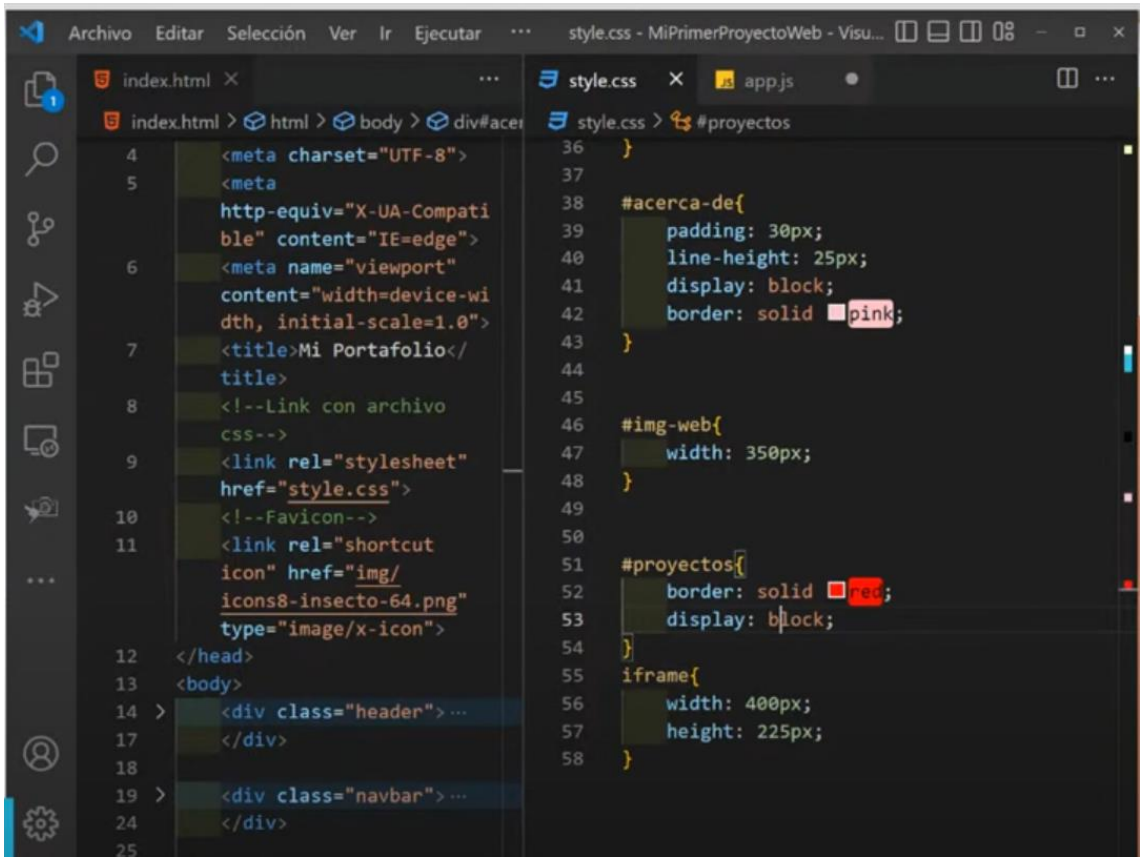
<h3>Episodio 1</h3>
<hr>
<iframe src="https://www.youtube.com/embed/crlrJt5Cjzc" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

<h3>Episodio 2</h3>

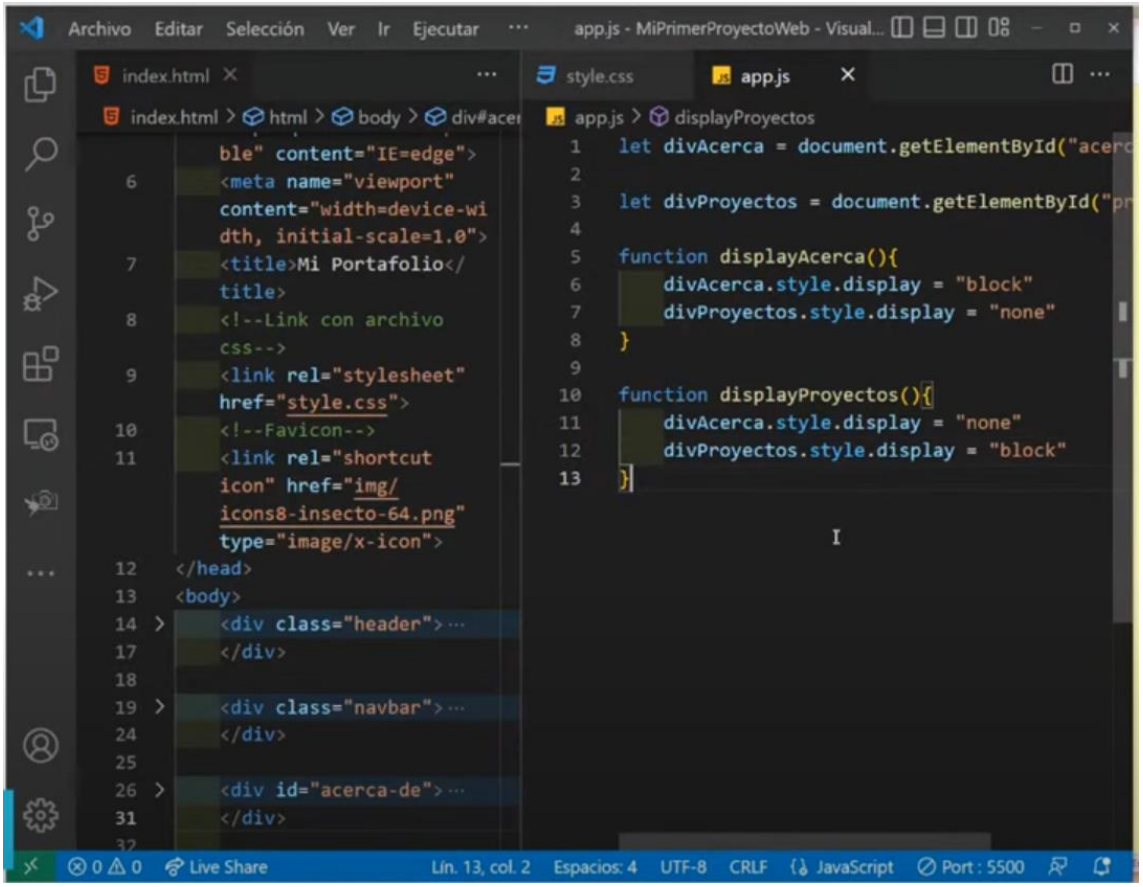
<iframe src="https://www.youtube.com/embed/M1UByWBFO2Q" title="YouTube video player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture"
button:hover{
color: black;
}
.acerca-de{
padding: 30px;
line-height: 25px;
}
#img-web{
width: 350px;
}
.proyecto{
border: solid red;
}
iframe{
width: 400px;
height: 225px;
}
Live Share
Lín. 37, col. 17
Espacios: 4
UTF-8
CRLF
HTML
Port : 5500


Archivo Editar Selección Ver Ir Ejecutar
style.css - MiPrimerProyectoWeb - Visua...
index.html
style.css
app.js
index.html > html > body > div#proyectos
style.css > #proyectos
<link rel="stylesheet" href="style.css">
<!--Favicon-->
<link rel="shortcut icon" href="img/icons8-insecto-64.png" type="image/x-icon">
</head>
<body>

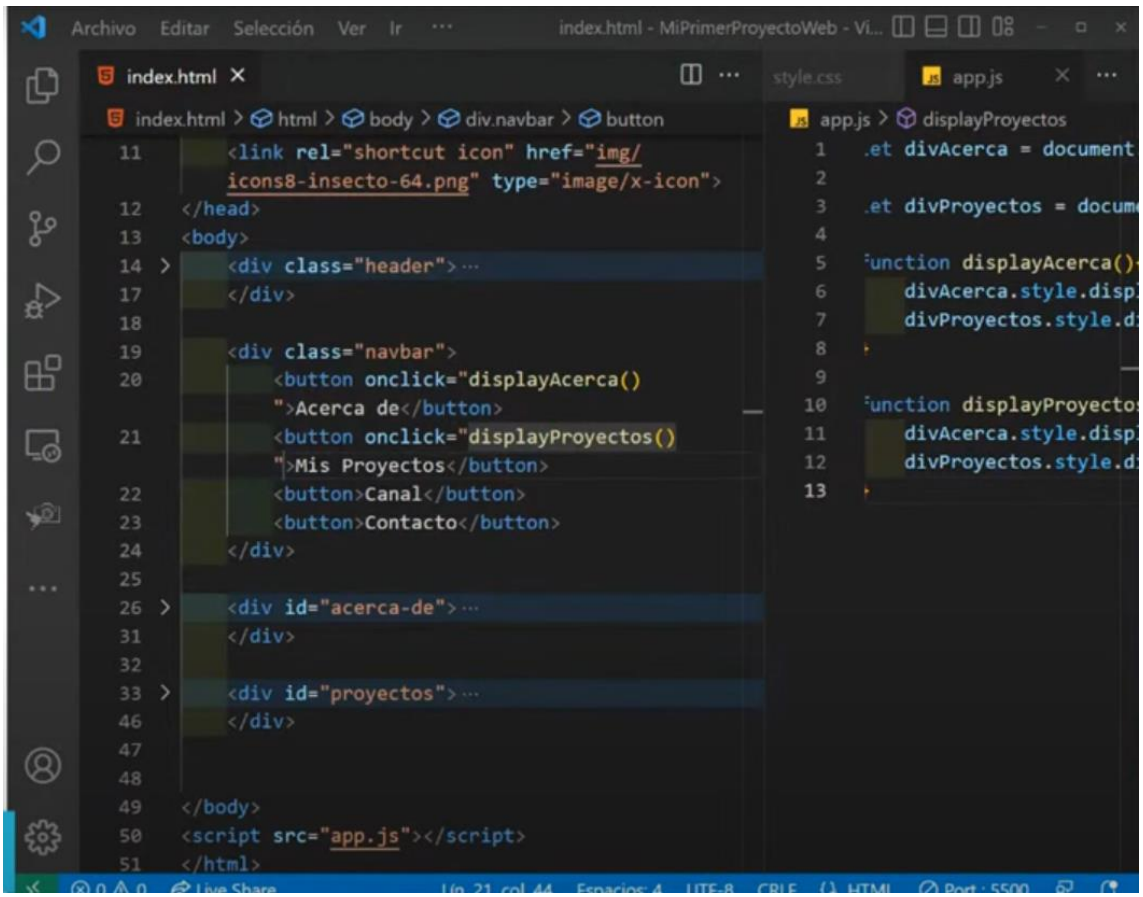
</body>
<script src="app.js"></script>
</html>
button:hover{ color: black; }
#acerca-de{ padding: 30px; line-height: 25px; display: none; }
#img-web{ width: 350px; }
#proyectos{ display: none; }
iframe{ width: 400px; height: 225px; }
Live Share
Lín. 51, col. 19
Espacios: 4
UTF-8
CRLF
CSS
Port : 5500


Archivo Editar Selección Ver Ir Ejecutar
style.css - MiPrimerProyectoWeb - Visu...
index.html
style.css
app.js
index.html > html > body > div#acer
style.css > #proyectos
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Mi Portafolio</title>
<!--Link con archivo css-->
<link rel="stylesheet" href="style.css">
<!--Favicon-->
<link rel="shortcut icon" href="img/icons8-insecto-64.png" type="image/x-icon">
</head>
<body>

#acerca-de{ padding: 30px; line-height: 25px; display: block; border: solid pink; }
#img-web{ width: 350px; }
#proyectos{ border: solid red; display: block; }
iframe{ width: 400px; height: 225px; }

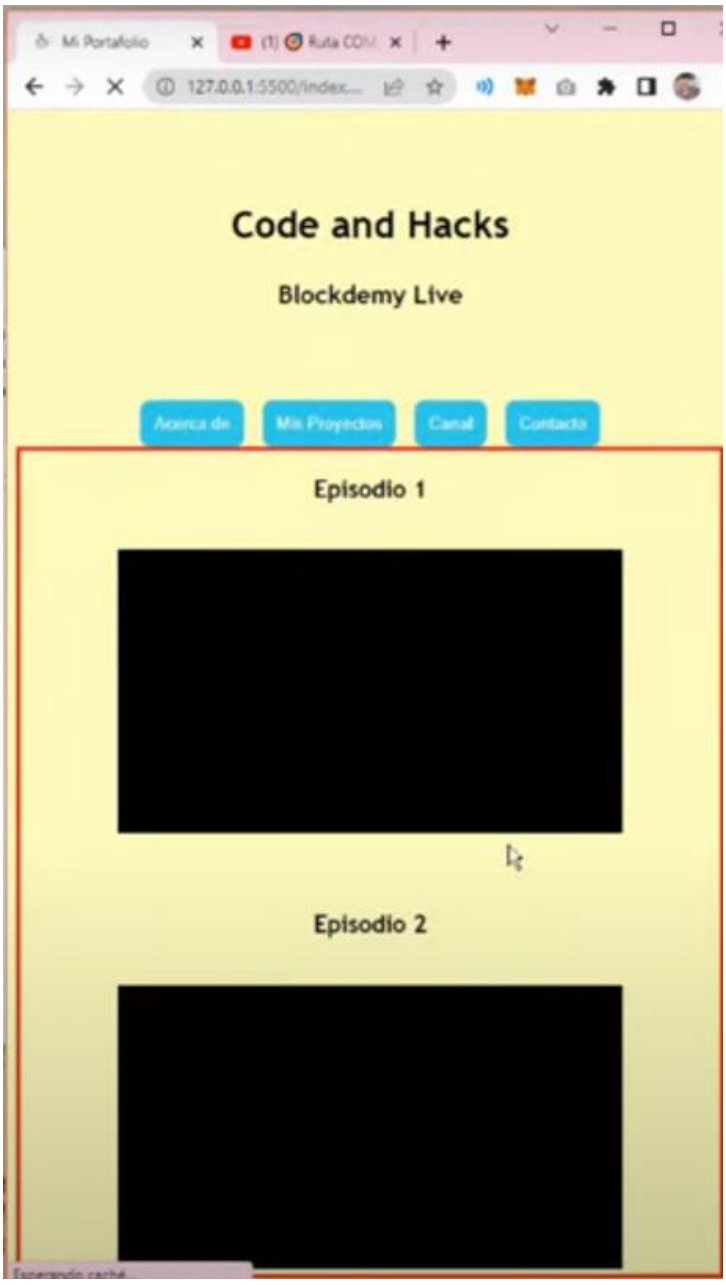

Archivo Editar Selección Ver Ir Ejecutar
app.js - MiPrimerProyectoWeb - Visual...
index.html
style.css
app.js
index.html > html > body > div#acer
app.js > displayProyectos
ble" content="IE=edge">
<meta name="viewport"
content="width=device-wi
dth, initial-scale=1.0">
<title>Mi Portafolio</
title>
<!--Link con archivo
css-->
<link rel="stylesheet"
href="style.css">
<!--Favicon-->
<link rel="shortcut
icon" href="img/
icons8-insecto-64.png"
type="image/x-icon">
</head>
<body>

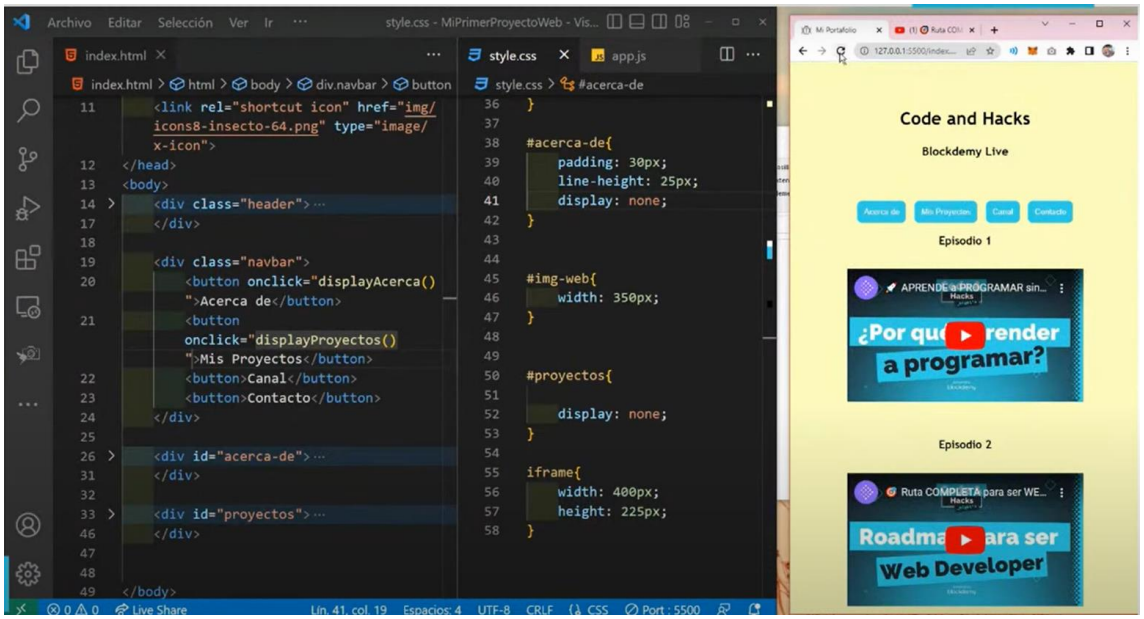
let divAcerca = document.getElementById("acerc
let divProyectos = document.getElementById("pr
function displayAcerca(){
divAcerca.style.display = "block"
divProyectos.style.display = "none"
}
function displayProyectos(){
divAcerca.style.display = "none"
divProyectos.style.display = "block"
}
0 0
Live Share
Lín. 13, col. 2
Espacios: 4
UTF-8
CRLF
JavaScript
Port : 5500


Archivo Editar Selección Ver Ir
index.html - MiPrimerProyectoWeb - Vi...
index.html
style.css
app.js
index.html > html > body > div.navbar > button
app.js > displayProyectos
<link rel="shortcut icon" href="img/
icons8-insecto-64.png" type="image/x-icon">
</head>
<body>

<button onclick="displayAcerca()
">Acerca de</button>
<button onclick="displayProyectos()
">Mis Proyectos</button>
<button>Canal</button>
<button>Contacto</button>

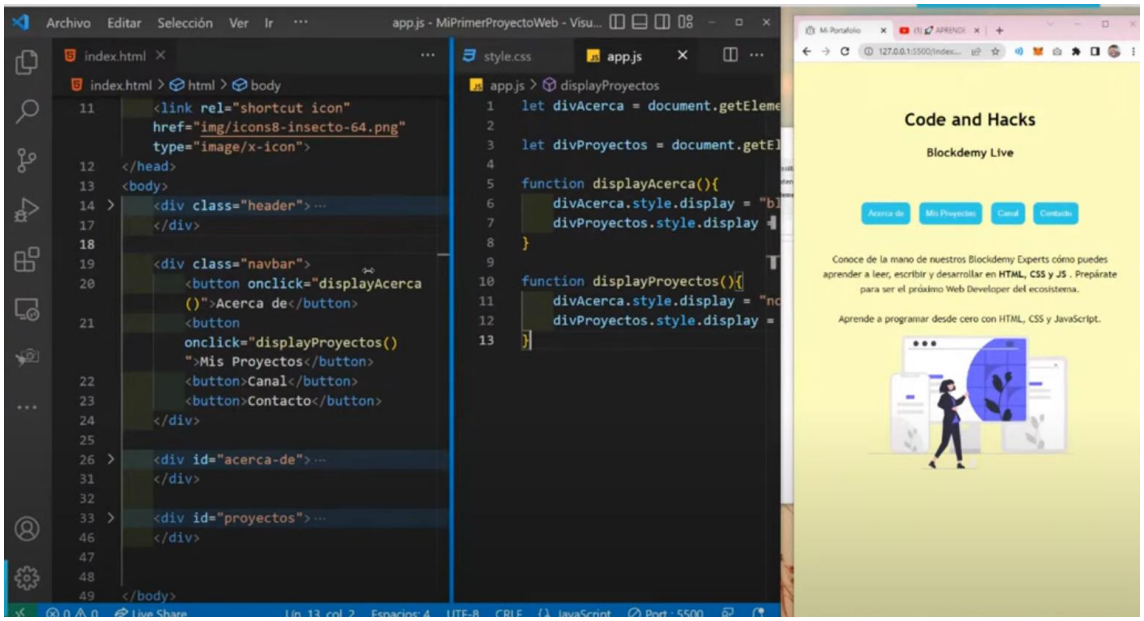
</body>
<script src="app.js"></script>
</html>
.et divAcerca = document
.et divProyectos = docum
unction displayAcerca()
divAcerca.style.displ
divProyectos.style.d
unction displayProyecto
divAcerca.style.displ
divProyectos.style.d


Mi Portafolio
127.0.0.1:5500/index...
Code and Hacks
Blockdemy Live
Acerca de
Mis Proyectos
Canal
Contacto
Episodio 1
Episodio 2


Archivo Editar Selección Ver Ir
style.css - MiPrimerProyectoWeb - Vis...
index.html
style.css
app.js
index.html > html > body > div.navbar > button
style.css > #acerca-de
<link rel="shortcut icon" href="img/icons8-insecto-64.png" type="image/x-icon">
</head>
<body>

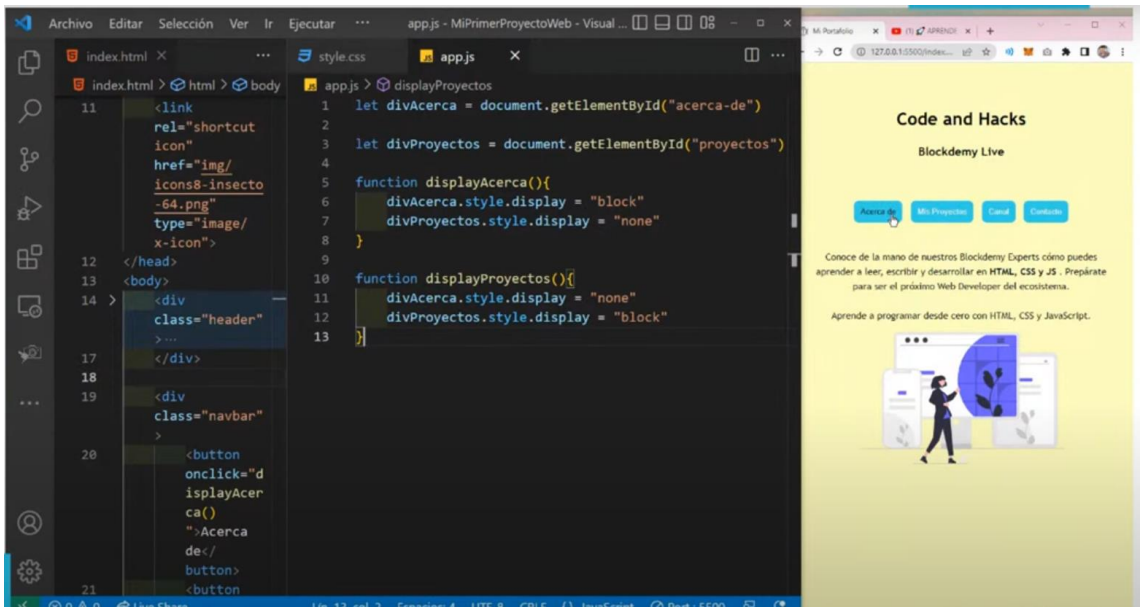
<button onclick="displayAcerca()">Acerca de</button>
<button onclick="displayProyectos()">Mis Proyectos</button>
<button>Canal</button>
<button>Contacto</button>

</body>
}
#acerca-de{
padding: 30px;
line-height: 25px;
display: none;
}
#img-web{
width: 350px;
}
#proyectos{
display: none;
}
iframe{
width: 400px;
height: 225px;
}
Code and Hacks
Blockdemy Live
Episodio 1
Episodio 2
Web Developer


Archivo Editar Selección Ver Ir
app.js - MiPrimerProyectoWeb - Visu...
index.html
style.css
app.js
index.html > html > body
app.js > displayProyectos
<link rel="shortcut icon" href="img/icons8-insecto-64.png" type="image/x-icon">
</head>
<body>

<button onclick="displayAcerca()">Acerca de</button>
<button onclick="displayProyectos()">Mis Proyectos</button>
<button>Canal</button>
<button>Contacto</button>

</body>
let divAcerca = document.getEleme
let divProyectos = document.getEl
function displayAcerca(){
divAcerca.style.display = "bl
divProyectos.style.display
}
function displayProyectos(){
divAcerca.style.display = "no
divProyectos.style.display =
}
Code and Hacks
Blockdemy Live
Conoce de la mano de nuestros Blockdemy Experts cómo puedes aprender a leer, escribir y desarrollar en HTML, CSS y JS . Prepárate para ser el próximo Web Developer del ecosisteema.
Aprende a programar desde cero con HTML, CSS y JavaScript.


Archivo Editar Selección Ver Ir Ejecutar
app.js - MiPrimerProyectoWeb - Visual ...
index.html
style.css
app.js
index.html > html > body
app.js > displayProyectos
<link rel="shortcut icon" href="img/icons8-insecto-64.png" type="image/x-icon">
</head>
<body>
...

<button onclick="displayAcerca()">Acerca de</button>
<button
let divAcerca = document.getElementById("acerca-de")
let divProyectos = document.getElementById("proyectos")
function displayAcerca(){
divAcerca.style.display = "block"
divProyectos.style.display = "none"
}
function displayProyectos(){
divAcerca.style.display = "none"
divProyectos.style.display = "block"
}
Code and Hacks
Blockdemy Live
Acerca de
Mis Proyectos
Canal
Contacto
Conoce de la mano de nuestros Blockdemy Experts cómo puedes aprender a leer, escribir y desarrollar en HTML, CSS y JS . Prepárate para ser el próximo Web Developer del ecosistema.
Aprende a programar desde cero con HTML, CSS y JavaScript.