sustantiva
CON•SENTIDO

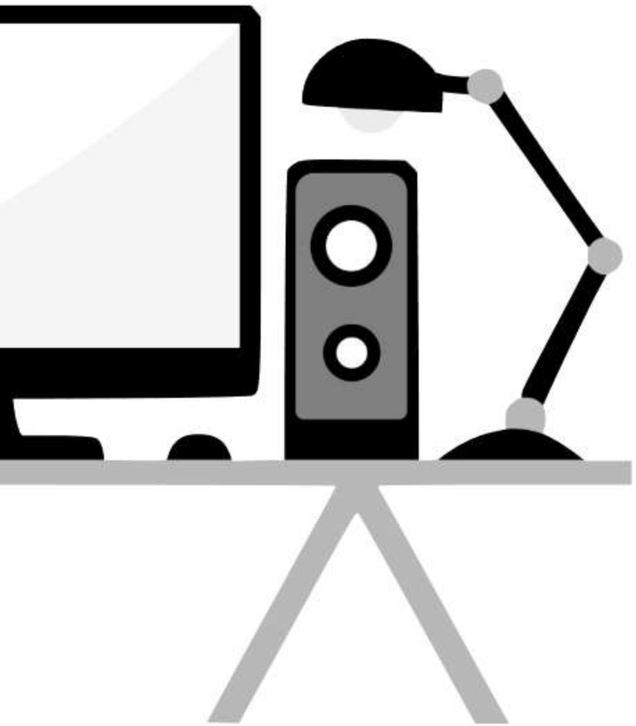
ARREGLOS Y CICLOS

Lección 03

Competencia General

Utilizar estructuras de tipo arreglo y sentencias iterativas para el control del flujo de un algoritmo que resuelve un problema simple acorde al lenguaje JavaScript.

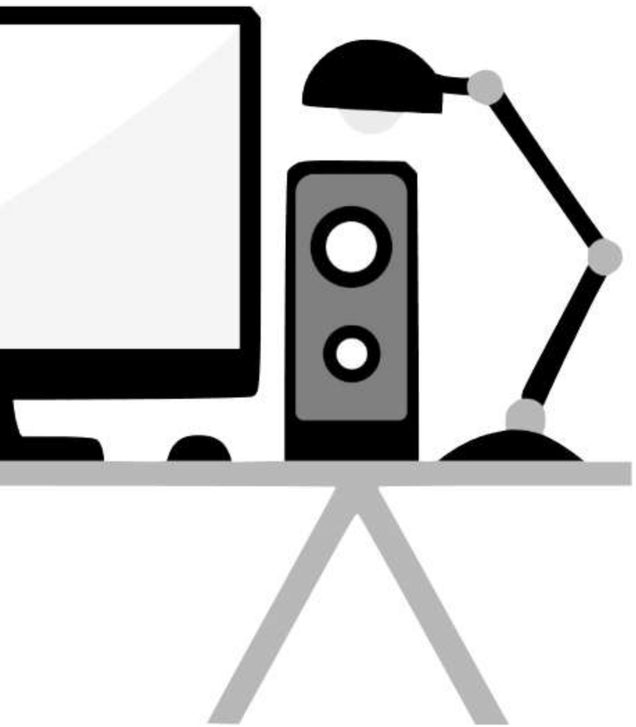
Contenidos



Arreglos

1. ¿Qué es un arreglo? Y cuando lo necesitamos
2. Crear arreglos
3. Acceder a un elemento de un arreglo
4. Contar los elementos de un arreglo
5. Iteraciones sobre un arreglo
6. Insertar y borrar elementos de un arreglo
7. Álgebra con arreglos
8. Matrices y arreglos asociativos

Contenidos



Ciclos iterativos

1. Para qué sirven los ciclos
2. Ciclo while
3. Asignaciones en un ciclo
4. Sumatorias
5. Scope de una variable en un ciclo
6. Instrucción do/while
7. Instrucción for
8. Ciclos anidados
9. Combinación de ciclos con if/else

Estilos, convenciones y buenas prácticas de codificación

1. El concepto código limpio
2. Por qué seguir una guía y convenciones
3. Convenciones y buenas prácticas

¿Qué es un arreglo?

Un arreglo es una estructura de datos que permite organizar información, esta queda almacenada en una colección finita y ordenada de elementos.

Los arreglos nos ayudan a coleccionar datos y almacenarlos en una sola variable. Algunos ejemplos que requieren manipular gran cantidad de datos son: análisis de indicadores financieros, predicción de clima, cálculos en ingeniería, entre otros.

```
// Declarando un array
var arreglo;

// Inicializando
arreglo = [“Dato1", “Dato2",
“Dato3"];
```

Creando arreglos

Para crear un arreglo debemos seguir las mismas **reglas de variables** establecidas en la lección anterior, para ello, se declara la variable, luego, se inicializa con los datos separados por coma (,) dentro de un paréntesis cuadrado [ ]. Esta variable sigue las mismas **reglas de tipo** establecidas en la lección anterior, por tanto, un arreglo tipo **let** se puede inicializar vacío y luego almacenar información y un arreglo tipo **const** no es posible editarlo.

```
// Declarando un array
let datos;

// Inicializando
datos = [ ];
datos = [ “Dato1”, “Dato2” ];
const constantes = [3.1415, 2.113];
```

Acceder a datos de un arreglo

Una vez creado y almacenado los datos en el arreglo necesitamos acceder a cada elemento de ese arreglo para operar con ellos.
Para acceder a un elemento del arreglo JavaScript asigna una posición a cada elemento del arreglo, estas posiciones van desde la 0 hasta la n-1, siendo n el número total de elementos en el arreglo.

```
// Declarando un arreglo
let datos;

// Inicializando
datos = [ “Dato1”, “Dato2”, “Dato3” ];

// Accediendo al arreglo
datos[0]; // devuelve “Dato1”
datos[1]; // devuelve “Dato2”
datos[2]; // devuelve “Dato3”
```

Contar elementos de un arreglo

JavaScript incluye algunas funciones para operar sobre arreglos.

Podemos contar el total de elementos en un arreglo con la función *length.*

Para saber en qué posición se encuentra un elemento podemos usar lastIndexOf(“Dato”);

```
// Declarando un arreglo
let datos;

// Inicializando
datos = [ “Dato1”, “Dato2”, “Dato3” ];

// Operaciones sobre arreglos
datos.length; // devuelve 3
datos.lastIndexOf(“Dato2”); // devuelve 1
datos.lastIndexOf(“Dato3”); // devuelve 2
```

Iteraciones sobre un arreglo

Una iteración es una secuencia de instrucciones repetitivas, la cual se ejecuta mientras se cumpla una condición dada.

La forma de construir estas iteraciones depende solo de la condición. Si la condición es repetir un número de veces una instrucción, entonces ocupamos **for.** Por el contrario si la condición no depende del número de repeticiones, entonces ocupamos **while.**

⚠ **Precaución con bucles,** asegúrese de que la condición en un bucle siempre llegue a término, de otra forma el bucle será infinito.

```
// Declarando
let datos, dato, contador;

// Inicializando
datos = [ “Dato1”, “Dato2”, “Dato3” ];
dato = “”;
contador = 0;
// Operaciones for
for (let i = 0; i < 3; i++){
  console.log(datos[i]);
} // retorna los 3 datos del arreglo

// Operaciones while
while (dato != "Dato2"){
  dato = datos[contador];
  contador++;
} // el ciclo termina cuando dato = “Dato2”
```

Insertar y borrar elementos de un arreglo

Para insertar datos en un arreglo, podemos insertar en una posición determinada, esto insertará el dato o modificará lo que tenga dicha posición o podemos insertar un dato al final del arreglo mediante la función ***push***(“Dato”)

```
// Declarando
let datos;

// Inicializando
datos = [ “Dato1”, “Dato2”, “Dato3” ];

// Insertando en una posición determinada
datos[4] = “Dato5”;
console.log(datos); // retorna [ “Dato1”, “Dato2”,
“Dato3”, undefined, “Dato5” ];

datos.push(“Dato6”);
console.log(datos); // retorna [ “Dato1”, “Dato2”,
“Dato3”, undefined, “Dato5”, “Dato6” ];
```

Insertar y borrar elementos de un arreglo

Para eliminar elementos de un arreglo existen distintas formas dependiendo de la ubicación del elemento a eliminar:

- Eliminar el primer elemento **shift()**
- Eliminar el último elemento **pop()**
- Eliminar un elemento según su índice **splice(inicio, nº de elementos)**
- Vaciar un arreglo con **length**

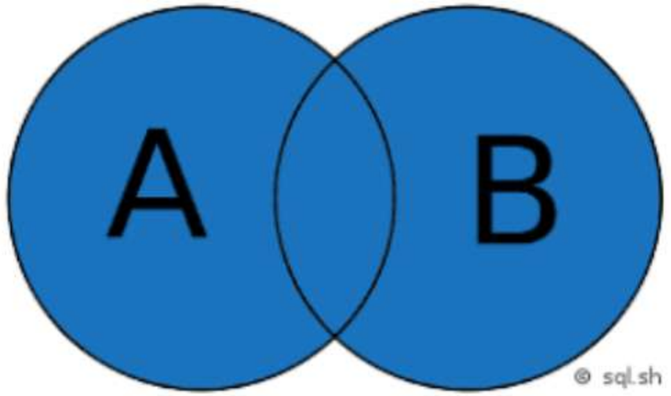
```
// Declarando
let datos;

// Inicializando
datos = [ “Dato1”,“Dato2”,“Dato3”,“Dato4”,“Dato5” ];

// Eliminando elementos
datos.shift();
console.log(datos); //retorna
[“Dato2”,“Dato3”,“Dato4”,“Dato5”];

datos.pop();
console.log(datos); //retorna [“Dato2”,“Dato3”,“Dato4”];

datos.splice(1,1);
console.log(datos); //retorna [“Dato2”,“Dato4”];

datos.length = 0; //datos = [ ];
```

Algebra sobre arreglos

Unión

La unión de dos arreglos forma un tercer arreglo que contiene a los elementos del primer y segundo arreglo.

Si existen elementos repetidos estos se incluyen solo 1 vez



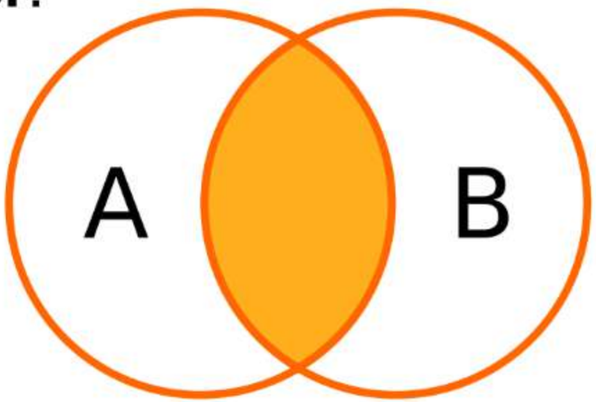

```
// Declarando
let arr1;
let arr2;
let arr3;

// Inicializando
arr1 = [ “Dato1”,“Dato2”];
arr2 = [ “Dato2”,“Dato3”];

// unión
arr3 = [...new Set([...arr1,...arr2])];
console.log(arr3); //retorna
[“Dato1”,“Dato2”,“Dato3”];
```

Algebra sobre arreglos

Intersección

La intersección de dos arreglos genera un tercer arreglo con los elementos repetidos del primer y segundo arreglo.

JavaScript no tiene de forma nativa una función para intersección, es por ello que hacemos uso de la función **filter**.



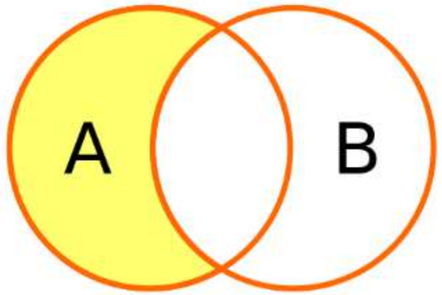

```
// Declarando
let arr1;
let arr2;
let arr3;

// Inicializando
arr1 = [ “Dato1”,“Dato2”];
arr2 = [ “Dato2”,“Dato3”];

// intersección
arr3 = arr1.filter(x => arr2.includes(x));
console.log(arr3); //retorna [“Dato2”];
```

Algebra sobre arreglos

Diferencia

La diferencia de dos arreglos genera un tercer arreglo con los elementos de un arreglo que no se repiten en el otro arreglo

JavaScript no tiene de forma nativa una función para diferencia, es por ello que hacemos uso de la función **filter**.




```
// Declarando
let arr1;
let arr2;
let arr3;

// Inicializando
arr1 = [ “Dato1”,“Dato2”];
arr2 = [ “Dato2”,“Dato3”];

// intersección
arr3 = arr1.filter(x => !arr2.includes(x));
console.log(arr3); //retorna [“Dato1”];
```

Algebra sobre arreglos

Concatenación

La concatenación de dos arreglos formará un tercer arreglo con los elementos del primer y segundo arreglo.

La concatenación no discrimina elementos repetidos, por ende si existen repetidos estos se agregan de igual forma al tercer arreglo.

```
// Declarando
let arr1;
let arr2;
let arr3;

// Inicializando
arr1 = [ “Dato1”,“Dato2”];
arr2 = [ “Dato2”,“Dato3”];

// intersección
arr3 = arr1.concat(arr2);
console.log(arr3); //retorna [“Dato1”];
```

Matrices y arreglos asociativos

Las matrices son arreglos de n dimensiones, por ende, esto implica que un arreglo contiene una serie de n elementos y cada elemento es en si una matriz.

Podemos representar la siguiente matriz mediante código:

```
// Declarando
let arr1;

// Inicializando
arr1 = [ [3,1] , [2,1] ];

// Desplegando matriz
console.log( arr1[0][0] ) // Muestra 3
console.log( arr1[0][1] ) // Muestra 1
console.log( arr1[1][0] ) // Muestra 2
console.log( arr1[1][1] ) // Muestra 1
```

Matrices y arreglos asociativos

Un arreglo asociativo se refiere a una matriz bidimensional donde cada elemento incluye su descripción.

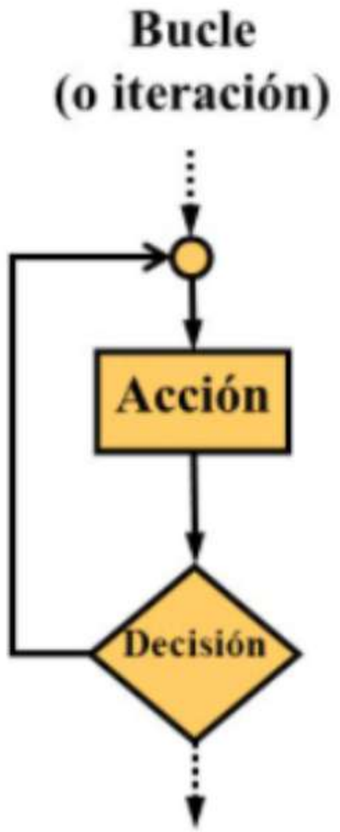
Para hacer uso de un arreglo asociativo debemos declarar el arreglo, luego asociar cada propiedad de este mediante una asignación “clave”:”valor”.

Por último para acceder a esa propiedad, llamamos al arreglo.propiedad

```
// Declarando
let auto;

// Inicializando
auto = [];
auto[“marca”] = “BMW”;
auto[“color”] = “Azul”;
auto[“patente”] = “XM ZK 92”;

// Desplegando
console.log(auto[“marca”]) // Muestra “BMW”
console.log(auto[“color”]) // Muestra “Azul”
console.log(auto[“patente”]) // Muestra “XM ZK 92”
```

Para qué sirven los ciclos

Un ciclo es un bloque de programación con instrucciones de código que se ejecutan repetidas veces, esta repetición puede estar condicionada a una sentencia a una expresión o a una cantidad de repeticiones.



JavaScript posee varias formas de ejecutar ciclos:

- For
- While
- Do while
- For-in
- For-of

Ciclo while

Un ciclo while ejecuta sus instrucciones mientras la condición de parada sea verdadera.

En el ejemplo el bucle incrementa en 1 la variable i y añade ese valor a x.

```
// Declarando
let i, x;

// Inicializando
i = 0;
x = 0;

// Ciclo while
while (i < 10) {
  i++; // incrementa en 1 la variable i
  x += i; // acumulador de la variable i
}
```

- En la primera iteración i=1, x=1
- En la segunda iteración i=2, x=3
- En la tercera iteración i=3, x=6
- …

El ciclo se termina cuando la variable i tiene el valor de 10, puesto que la condición es falsa en ese instante.

Asignaciones en un ciclo

En un ciclo podemos realizar asignaciones de variables, estas actualizarán el valor de la variable. En el ejemplo visto, se realiza una asignación a la variable i, x.

Podemos realizar todo tipo de asignaciones ya sean operaciones aritméticas o asignaciones de texto.

Las asignaciones también poseen métodos abreviados como en la siguiente tabla.

Operador	Asignación
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`

Sumatorias en un ciclo

Usando los operadores de asignación en un ciclo podemos realizar todo tipo de sumatorias y almacenar el resultado en un ciclo

Por ejemplo la siguiente sumatoria en código sería:

$$\sum_{n=0}^{5} i$$

```
// Declarando
let i, n;

// Inicializando
i = 0;
x = 0;

// Ciclo while
while (n < 5) {
  n++; // incrementa en 1 la variable i
  i += n; // acumulador de la variable i
}
console.log(i) // retorna 15 (1+2+3+4+5)
```

Scope de una variable en un ciclo

Las variables declaradas en el bloque interior del ciclo solo estarán disponibles en el ciclo.

Las variables declaradas fuera del ciclo, se pueden utilizar dentro o fuera del ciclo.

```
let i; // Variables disponibles fuera y dentro del ciclo.


// Ciclo while
while (condicion) {
   let x; // variable de uso local solo disponible en el ciclo.

}
```

Instrucción do while

El ciclo *do while* es una variante del ciclo *while*. Su particularidad se centra en primero realizar una iteración (la primera) y luego evaluar la condición de parada.

```
// Declarando
let i;

// Inicializando
i = 0;

// Ciclo do
do {
  i += 1; // acumulador de la variable i
  console.log(i); // imprime 1 2 3 4 5 un numero en
          cada iteracion
} while ( i < 5);
```

Instrucción for

Un bucle for se repite hasta que la condición de parada sea false.

Este bucle requiere que se introduzcan 3 parámetros.

- Una expresión inicial
- Una condición de parada
- Una expresión de incremento

```
// Ciclo for
for (expresión inicial ; condición ; incremento) {
  //  sentencias que se ejecutan si la condición es
  true
}
```

Ciclos anidados

Un ciclo anidado consiste en construir un ciclo al interior de otro ciclo. Este tipo de instrucción nos ayuda a acceder a arreglos que se encuentran al interior de otros arreglos, por ejemplo acceder a todos los elementos de una matriz.

```
//Ciclos anidados
for (let i=0; i<10 ; i++){
    for (let j=0; j<10 ;j++) {
      //sentencias
    }
}
```

El bucle exterior instancia una variable auxiliar **i** la cual estará disponible para usar en los dos bucles. Por el contrario la variable **j** solo está disponible en el bucle interior.

Combinar ciclos con sentencias if else

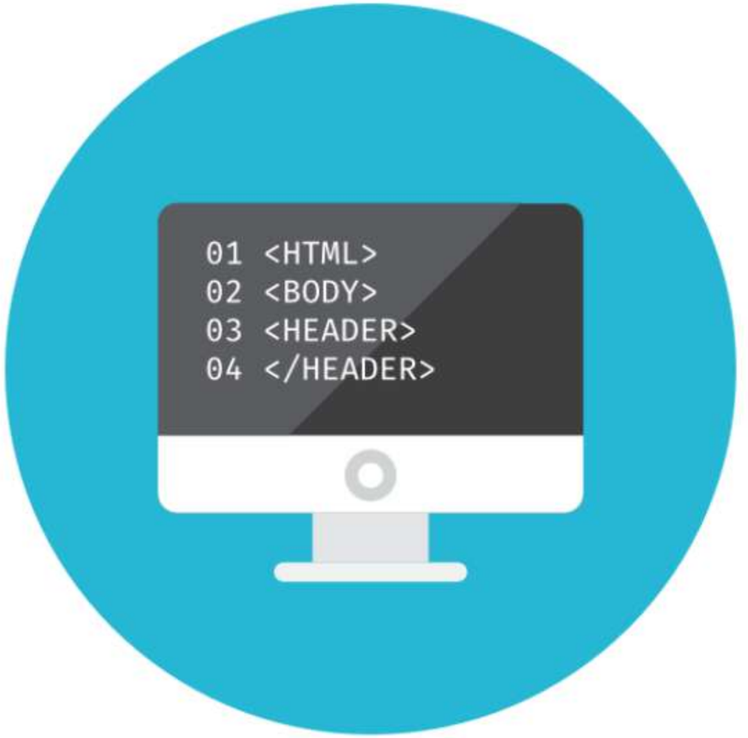
Es posible cortar la ejecución de un ciclo antes de evaluar la condición de parada. Para ello hacemos uso de ***break;*** Esto puede estar en una condición al interior del ciclo

```
//Ciclos anidados
let variable = 0;
for (let i=0; i<10 ; i++){
  if (i == 2) {
    break;
  }
}
```

El concepto código limpio

Cuando hablamos de código limpio, nos referimos a estilos de desarrollo centrado en entender el código, en concreto, un código que sea fácil de leer, escribir y mantener.

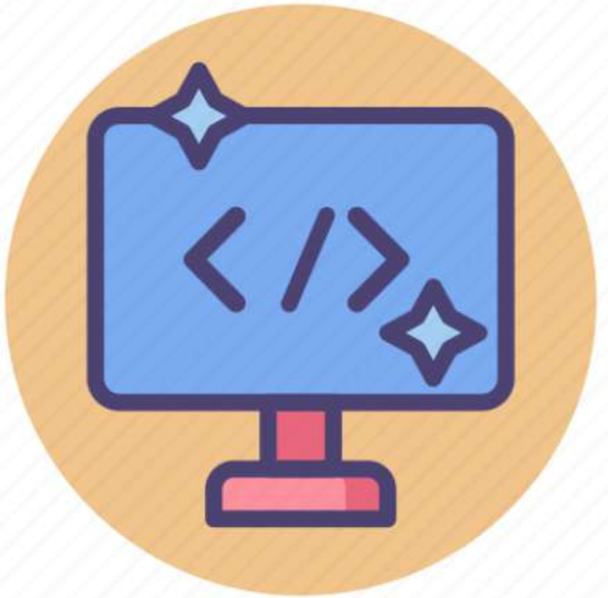
"Son los buenos programadores los que escriben códigos que los humanos pueden entender" – Marin Fowler



El concepto código limpio

Ocho razones por las que mantener un código limpio es importante:

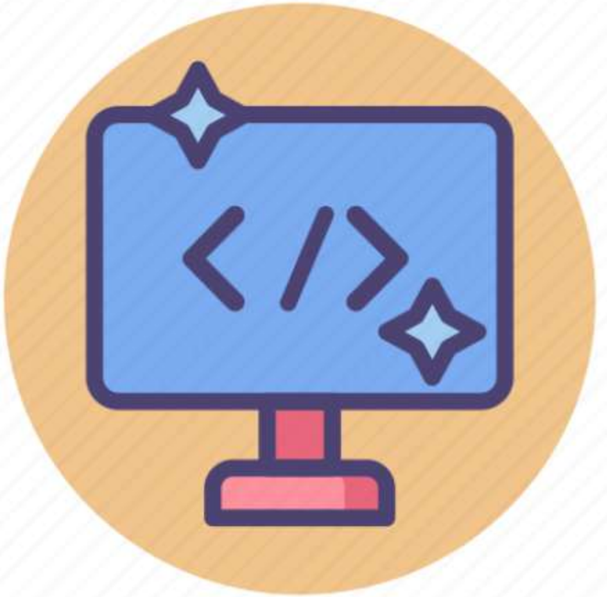
1. Limpieza.
2. Mejores prácticas.
3. Lógica
4. Mantenimiento
5. Fácil de testeo
6. Simplicidad
7. Consistencia de la información
8. Ahorro de tiempo



Por qué seguir una guía de buenas prácticas

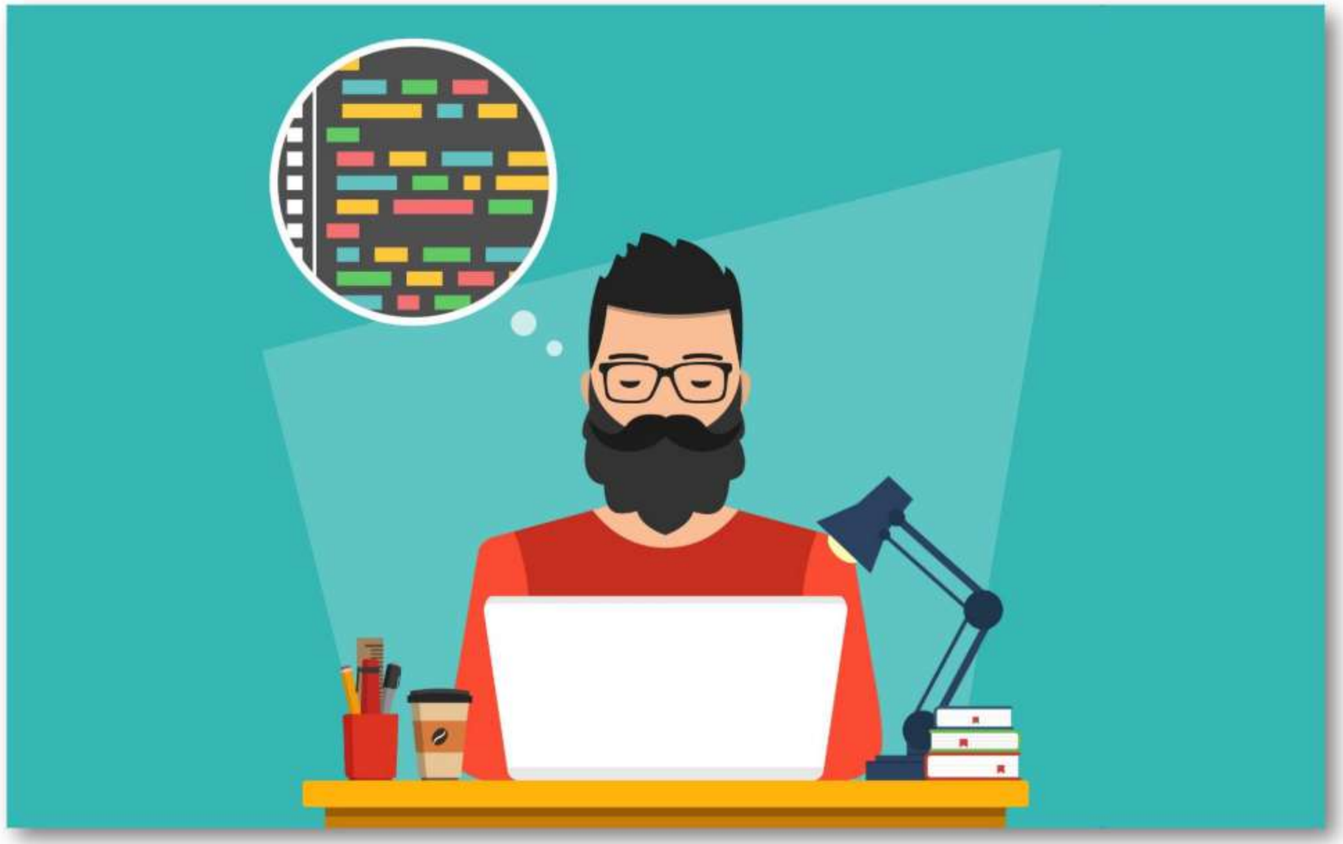
Al seguir una guía de buenas prácticas obtenemos como resultado un código limpio, reutilizable y escalable, el cual nos ayudará a disminuir el tiempo que dedicamos a sortear errores en las diferentes etapas de un proyecto.

Las buenas prácticas también nos ayudan a trabajar de manera colaborativa.



Convenciones y buenas prácticas

- Evita repetir código o estructuras que realizan acciones similares
- Reduce la dependencia de HTML con JavaScript (onclick en HTML)
- Elimina código innecesario
- Evita cálculos en estructuras de ciclo
- Realiza pruebas de rendimiento a tus algoritmos



Ejercicio práctico

Realiza un programa donde el usuario introduzca el mes (1 a 12) y la respuesta sea la cantidad de días que tiene dicho mes. (Utiliza arreglos para almacenar los meses, realiza todas las condicionales que estipules necesarias y verifica todas las posibles entradas de datos)


sustantiva
CON•SENTIDO