INTRODUCCIÓN A JAVASCRIPT

2º DAW - Desarrollo Web en Entorno Cliente

¿Qué es JavaScript?

JavaScript es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas.

Una página web dinámica es aquella que incorpora efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

A pesar de su nombre, JavaScript no guarda ninguna relación directa con el lenguaje de programación Java. Legalmente, JavaScript es una marca registrada de la empresa Sun Microsystems, como se puede ver en http://www.sun.com/suntrademarks/.

Fundamentos - Variables

Una variable es un “almacén con un nombre” para guardar datos. Podemos usar variables para almacenar golosinas, visitantes, y otros datos. Para generar una variable en JavaScript, se usa la palabra clave **let**.

La siguiente declaración genera (en otras palabras: declara o define):

1. una variable con el nombre “message”.

2. Introducir datos en ella al utilizar el operador de asignación =.

3. La cadena ahora está almacenada en el área de la memoria asociada con la variable. La podemos acceder utilizando el nombre de la variable.

```
1  let message;
2  message = 'Hola!';
3
4  alert(message); // muestra el contenido de la variable
```

Pero por el bien de la legibilidad se recomienda una línea por variable:

```
1  let user = 'John';
2  let age = 25;
3  let message = 'Hola';
```

```
1  let user = 'John',
2    age = 25,
3    message = 'Hola';
```

```
1  let user = 'John'
2    , age = 25
3    , message = 'Hola';
```

NOTA: En scripts más viejos se puede encontrar la palabra clave VAR en lugar de LET.

Ejemplo de cómo mover el contenido de variables entre las mismas:

```
let hello = 'Hola mundo!';

let message;

// copia 'Hola mundo' de hello a message
message = hello;

// Ahora, ambas variables contienen los mismos datos
alert(hello); // Hola mundo!
alert(message); // Hola mundo!
```

Nombramiento de variables

Existen dos limitaciones de nombre de variables en JavaScript:

a. El nombre únicamente puede incluir letras, dígitos, o los símbolos $ y _.

b. El primer carácter no puede ser un dígito.

NOTA1: Es interesante notar que el símbolo del dólar '$' y el guion bajo '_' también se utilizan en nombres. Son símbolos comunes, tal como las letras, sin ningún significado especial.

NOTA2: Dos variables con nombres manzana y MANZANA son variables distintas.

Hay una lista de palabras reservadas, las cuales no pueden ser utilizadas como nombre de variable porque el lenguaje en sí las utiliza. Por ejemplo: **let, class, return, y function** están reservadas.

Constantes

Para declarar una variable constante (inmutable) usa “**const**” en vez de “let”. Las variables declaradas utilizando **const** se llaman “constantes”. No pueden ser alteradas. Al intentarlo causaría un error:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // ¡error, no se puede reasignar la constante!
```

Constantes Mayúsculas

Existe una práctica utilizada ampliamente de utilizar constantes como aliases de valores difíciles-de-recordar y que se conocen previo a la ejecución. Tales constantes se nombran utilizando letras mayúsculas y guiones bajos.

Por ejemplo, creemos constantes para los colores en el formato “web” (hexadecimal):

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...cuando debemos elegir un color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Ventajas:

1. COLOR_ORANGE es mucho más fácil de recordar que "#FF7F00".
2. Es mucho más fácil escribir mal "#FF7F00" que COLOR_ORANGE.

Al leer el código, COLOR_ORANGE tiene mucho más significado que #FF7F00.

RESUMEN

- Podemos declarar variables para almacenar datos al utilizar las palabras clave var, let, o const.
- let – es la forma moderna de declaración de una variable.
- var – es la declaración de variable de vieja escuela. Normalmente no lo utilizamos en absoluto. Cubriremos sus sutiles diferencias con let en el capítulo La vieja "var", por si lo necesitaras.
- const – es como let, pero el valor de la variable no puede ser alterado.

TAREAS

1. Trabajando con variables

Declara dos variables: `admin` y `name`.

Asigna el valor `"Javier"` a `name`.

Copia el valor de `name` a `admin`.

Muestra el valor de `admin` usando `alert`.

2. Dando el nombre correcto

Crea una variable con el nombre de nuestro planeta. ¿Cómo nombrarías a dicha variable?

Crea una variable para almacenar el nombre del usuario actual de un sitio web. ¿Cómo nombrarías a dicha variable?

3. Constantes Mayúsculas

Examina el siguiente código:

```
const birthday = '18.04.1986';
const age = someCode(birthday);
```

Aquí tenemos una constante `birthday` para la fecha de cumpleaños, y la edad age, que también es constante.

age es calculada desde `birthday` con la ayuda de "cierto código" `someCode()`, que es una llamada a función que no hemos explicado aún (¡lo haremos pronto!); los detalles no importan aquí, el punto es que age se calcula de alguna forma basándose en `birthday`.

¿Sería correcto usar mayúsculas en `birthday`? ¿Y en age? ¿O en ambos?

```
const BIRTHDAY = '18.04.1982'; // ¿birthday en mayúsculas?
const AGE = someCode(BIRTHDAY); // ¿age en mayúsculas?
```

Constantes

Para declarar una variable constante (inmutable) usa “**const**” en vez de “let”. Las variables declaradas utilizando **const** se llaman “constantes”. No pueden ser alteradas. Al intentarlo causaría un error:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // ¡error, no se puede reasignar la constante!
```

Constantes Mayúsculas

Existe una práctica utilizada ampliamente de utilizar constantes como aliases de valores difíciles-de-recordar y que se conocen previo a la ejecución. Tales constantes se nombran utilizando letras mayúsculas y guiones bajos.

Por ejemplo, creemos constantes para los colores en el formato “web” (hexadecimal):

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...cuando debemos elegir un color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Ventajas:

1. COLOR_ORANGE es mucho más fácil de recordar que "#FF7F00".
2. Es mucho más fácil escribir mal "#FF7F00" que COLOR_ORANGE.

Al leer el código, COLOR_ORANGE tiene mucho más significado que #FF7F00.

RESUMEN

- Podemos declarar variables para almacenar datos al utilizar las palabras clave var, let, o const.
- let – es la forma moderna de declaración de una variable.
- var – es la declaración de variable de vieja escuela. Normalmente no lo utilizamos en absoluto. Cubriremos sus sutiles diferencias con let en el capítulo La vieja "var", por si lo necesitaras.
- const – es como let, pero el valor de la variable no puede ser alterado.

Tipos de datos

NUMBER

El tipo number representa tanto números enteros como de punto flotante. Hay muchas operaciones para números. Por ejemplo, multiplicación *, división /, suma +, resta -, y demás.

Además de los números comunes, existen los llamados “valores numéricos especiales” que también pertenecen a este tipo de datos: Infinity, -Infinity y NaN.

```
alert( 1 / 0 ); // Infinity
```

```
alert( Infinity ); // Infinity
```

```
alert( NaN + 1 ); // NaN
alert( 3 * NaN ); // NaN
alert( "not a number" / 2 - 1 ); // NaN
```

NOTA: Por lo tanto, si hay un NaN en alguna parte de una expresión matemática, se propaga a todo el resultado (con una única excepción: NaN ** 0 es 1).

BIGINT

En JavaScript, el tipo “number” no puede representar de forma segura valores enteros mayores que (253-1) (eso es 9007199254740991), o menor que -(253-1) para negativos.

Un valor BigInt se crea agregando n al final de un entero:

```
// la "n" al final significa que es un BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

NOTA: Problemas de compatibilidad → En este momento, BigInt está soportado por Firefox/Chrome/Edge/Safari, pero no por IE.

STRING

Un string en JavaScript es una cadena de caracteres y debe colocarse entre comillas. En JavaScript, hay 3 tipos de comillas.

a. Comillas dobles: "Hola".
b. Comillas simples: 'Hola'.
c. Backticks (comillas invertidas): `Hola`.

Las comillas dobles y simples son comillas “sencillas” (es decir, funcionan igual). No hay diferencia entre ellas en JavaScript.

Los backticks son comillas de “funcionalidad extendida”. Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en ${...}, por ejemplo:

```
let name = "John";

// incrustar una variable
alert( `Hola, ${name}!` ); // Hola, John!

// incrustar una expresión
alert( `el resultado es ${1 + 2}` ); //el resultado es 3
```

BOOLEAN (TIPO LÓGICO)

El tipo boolean tiene sólo dos valores posibles: true y false.

Este tipo se utiliza comúnmente para almacenar valores de sí/no: true significa “sí, correcto, verdadero”, y false significa “no, incorrecto, falso”.

Por ejemplo:

```
let nameFieldChecked = true; // sí, el campo name está marcado
let ageFieldChecked = false; // no, el campo age no está marcado
```

VALOR "NULL" (NULO)

El valor especial null no pertenece a ninguno de los tipos descritos anteriormente. Forma un tipo propio separado que contiene sólo el valor null:

```
let age = null;
```

En JavaScript, null no es una "referencia a un objeto inexistente" o un "puntero nulo" como en otros lenguajes. Es sólo un valor especial que representa "nada", "vacío" o "valor desconocido".

VALOR UNDEFINED

El valor especial undefined también se distingue. Hace un tipo propio, igual que null. El significado de undefined es "valor no asignado".

Si una variable es declarada, pero no asignada, entonces su valor es undefined, es posible asignar undefined a cualquier variable:

```
let age = 100;

// cambiando el valor a undefined
age = undefined;

alert(age); // "undefined"
```

NOTA: Normalmente, usamos null para asignar un valor "vacío" o "desconocido" a una variable, mientras undefined es un valor inicial reservado para cosas que no han sido asignadas.

EL OPERADOR TYPEOF

El operador typeof devuelve el tipo de dato del operando. Es útil cuando queremos procesar valores de diferentes tipos de forma diferente o simplemente queremos hacer una comprobación rápida. La llamada a typeof x devuelve una cadena con el nombre del tipo:

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Las últimas tres líneas pueden necesitar una explicación adicional:

a) **Math** es un objeto incorporado que proporciona operaciones matemáticas. Lo aprenderemos en el capítulo Números. Aquí sólo sirve como ejemplo de un objeto.
b) El resultado de **typeof null** es "object". Esto está oficialmente reconocido como un error de comportamiento de typeof que proviene de los primeros días de JavaScript y se mantiene por compatibilidad.
c) El resultado de **typeof alert** es "function" porque alert es una función. Las funciones pertenecen al tipo objeto. Pero typeof las trata de manera diferente, devolviendo function.

NOTA: typeof es un operador, no una función. Los paréntesis aquí no son parte del operador typeof. Son del tipo usado en agrupamiento matemático.

RESUMEN

- Hay 8 tipos básicos en JavaScript.
- Siete tipos de datos primitivos
  - number para números de cualquier tipo: enteros o de punto flotante, los enteros están limitados por $\pm(2^{53}-1)$.
  - bigint para números enteros de longitud arbitraria.
  - string para cadenas. Una cadena puede tener cero o más caracteres, no hay un tipo especial para un único carácter.
  - boolean para verdadero y falso: true/false.
  - null para valores desconocidos – un tipo independiente que tiene un solo valor nulo: null.
  - undefined para valores no asignados – un tipo independiente que tiene un único valor "indefinido": undefined.
  - symbol para identificadores únicos.
- Y un tipo de dato no primitivo:
  - object para estructuras de datos complejas.
- El operador typeof nos permite ver qué tipo está almacenado en una variable.
  - Dos formas: typeof x o typeof(x).
  - Devuelve una cadena con el nombre del tipo. Por ejemplo "string".
  - Para null devuelve "object": esto es un error en el lenguaje, en realidad no es un objeto.

TAREAS EN CLASE

```
¿Cuál es la salida del script?
let name = "Ilya";
alert( `Hola ${1}` ); // ?
alert( `Hola ${"name"}` ); // ?
alert( `Hola ${name}` ); // ?
```

Conversiones de tipo

La mayoría de las veces, los operadores y funciones convierten automáticamente los valores que se les pasan al tipo correcto. Esto es llamado “conversión de tipo”.

Por ejemplo, **alert** convierte automáticamente cualquier valor a string para mostrarlo. Las operaciones matemáticas convierten los valores a números. También hay casos donde necesitamos convertir de manera explícita un valor al tipo esperado.

ToString

La conversión a string ocurre cuando necesitamos la representación en forma de texto de un valor. Por ejemplo, alert(value) lo hace para mostrar el valor como texto. También podemos llamar a la función String(value) para convertir un valor a string:

```
1  let value = true;
2  alert(typeof value); // boolean
3
4  value = String(value); // ahora value es el string "true"
5  alert(typeof value); // string
```

ToNumber

La conversión numérica ocurre automáticamente en funciones matemáticas y expresiones. Por ejemplo, cuando se dividen valores no numéricos usando /:

```
alert( "6" / "2" ); // 3, los strings son convertidos a números
```

Podemos usar la función Number(value) para convertir de forma explícita un valor a un número:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // se convierte en 123

alert(typeof num); // number
```

Si el string no es un número válido, el resultado de la conversión será NaN. Por ejemplo:

```
let age = Number("un texto arbitrario en vez de un número");

alert(age); // NaN, conversión fallida
```

Reglas de conversión numérica:

Valor	Se convierte en...
`undefined`	`NaN`
`null`	`0`
`true and false`	`1` y `0`
`string`	Se eliminan los espacios (incluye espacios, tabs `\t`, saltos de línea `\n`, etc.) al inicio y final del texto. Si el string resultante es vacío, el resultado es `0`, en caso contrario el número es "leído" del string. Un error devuelve `NaN`.

Ejemplos:

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (error al leer un número en "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

NOTA: Ten en cuenta que null y undefined se comportan de distinta manera aquí: null se convierte en 0 mientras que undefined se convierte en NaN.

ToBoolean

Ocurre en operaciones lógicas (más adelante veremos test condicionales y otras cosas similares), pero también puede realizarse de forma explícita llamando a la función Boolean(value).

Las reglas de conversión:

- Los valores que son intuitivamente “vacíos”, como 0, "", null, undefined, y NaN, se convierten en false.
- Otros valores se convierten en true.

Por ejemplo:

```
1  alert( Boolean(1) ); // true
2  alert( Boolean(0) ); // false
3
4  alert( Boolean("hola") ); // true
5  alert( Boolean("") ); // false
```

NOTA: En JavaScript, un string no vacío es siempre true.

RESUMEN

- Las tres conversiones de tipo más usadas son a string, a número y a boolean.
- **ToString** – Ocurre cuando se muestra algo. Se puede realizar con String(value). La conversión a string es usualmente obvia para los valores primitivos.
- **ToNumber** – Ocurre en operaciones matemáticas. Se puede realizar con Number(value).
- **ToBoolean** – Ocurren en operaciones lógicas. Se puede realizar con Boolean(value). La mayoría de estas reglas son fáciles de entender y recordar. Las excepciones más notables donde la gente suele cometer errores son:
  - undefined es NaN como número, no 0.
  - "0" y textos que solo contienen espacios como " " son true como boolean.

Operadores básicos

Se van a revisar los siguientes términos: “unario”, “binario”, “operando”

- Un operando – es a lo que se aplican los operadores. Por ejemplo, en la multiplicación de 5 * 2 hay dos operandos: el operando izquierdo es 5 y el operando derecho es 2. A veces, la gente los llama “argumentos” en lugar de “operandos”.
- Un operador es unario si tiene un solo operando. Por ejemplo, la negación unaria - invierte el signo de un número:

```
let x = 1;

x = -x;
alert( x ); // -1, se aplicó negación unaria
```

- Un operador es binario si tiene dos operandos. El mismo negativo también existe en forma binaria:

```
let x = 1, y = 3;
alert( y - x ); // 2, binario negativo resta valores
```

Están soportadas las siguientes operaciones:

1. Suma “+”
2. Resta “-“
3. Multiplicación “*”
4. División “/”
5. Resto “%”
6. Exponenciación “**”

Se explicarán más ampliamente los 2 últimos operandos.

1. *Resto %*

El operador resto %, a pesar de su apariencia, no está relacionado con porcentajes. El resultado de a % b es el resto de la división entera de a por b. Por ejemplo:

```
1  alert( 5 % 2 ); // 1, es el resto de 5 dividido por 2
2  alert( 8 % 3 ); // 2, es el resto de 8 dividido por 3
3  alert( 8 % 4 ); // 0, es el resto de 8 dividido por 4
```

2. *Exponenciación ***

El operador exponenciación a ** b eleva a a la potencia de b. Por ejemplo:

```
1  alert( 2 ** 2 ); // 2² = 4
2  alert( 2 ** 3 ); // 2³ = 8
3  alert( 2 ** 4 ); // 2⁴ = 16
```

Por ejemplo, la raíz cuadrada es el exponente ½:

```
1  alert( 4 ** (1/2) ); // 2 (potencia de 1/2 es lo mismo que raíz cuadrada)
2  alert( 8 ** (1/3) ); // 2 (potencia de 1/3 es lo mismo que raíz cúbica)
```

Se van a revisar las características de los operadores de JavaScript.

Concatenación de cadenas con el binario

Si se aplica el + binario a una cadena, los une (concatena):

```
1  let s = "my" + "string";
2  alert(s); // mystring
```

A tener en cuenta, que si uno de los operandos es una cadena, el otro es convertido a una cadena también.

```
1  alert( '1' + 2 ); // "12"
2  alert( 2 + '1' ); // "21"
```

Aquí hay un ejemplo algo más complejo:

```
1  alert(2 + 2 + '1' ); // "41" y no "221"

1  alert('1' + 2 + 2); // "122", no es "14"
```

Aquí el primer operando es una cadena, el compilador trata los otros dos operandos como cadenas también. El 2 es concatenado a '1', entonces es como '1' + 2 = "12" y "12" + 2 = "122".

El binario + es el único operador que soporta cadenas en esa forma. Otros operadores matemáticos trabajan solamente con números y siempre convierten sus operandos a números.

Conversión numérica, unario "+"

La suma + existe en dos formas: la forma binaria que utilizamos arriba y la forma unaria. El unario suma o, en otras palabras, el operador suma + aplicado a un solo valor, no hace nada a los números. Pero si el operando no es un número, el unario suma lo convierte en un número. Por ejemplo:

```
// Sin efecto en números
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Convierte los no números
alert( +true ); // 1
alert( +"" );   // 0
```

El operador binario suma los agregaría como cadenas:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", el binario suma concatena las cadenas
```

Si queremos tratarlos como números, necesitamos convertirlos y luego sumarlos:

```
let apples = "2";
let oranges = "3";

// ambos valores convertidos a números antes del operador binario suma
alert( +apples + +oranges ); // 5

// la variante más larga
// alert( Number(apples) + Number(oranges) ); // 5
```

Precedencia del operador

Si una expresión tiene más de un operador, el orden de ejecución se define por su precedencia o, en otras palabras, el orden de prioridad predeterminado de los operadores. Los paréntesis anulan cualquier precedencia, por lo que si no estamos satisfechos con el orden predeterminado, podemos usarlos para cambiarlo. Por ejemplo, escriba (1 + 2) * 2.

Hay muchos operadores en JavaScript. Cada operador tiene un número de precedencia correspondiente. El que tiene el número más grande se ejecuta primero. Si la precedencia es la misma, el orden de ejecución es de izquierda a derecha.

Aquí hay un extracto de la tabla de precedencia (no necesita recordar esto, pero tenga en cuenta que los operadores unarios son más altos que el operador binario correspondiente):

Precedencia	Nombre	Signo
...	...	...
14	suma unaria	+
14	negación unaria	-
13	exponenciación	**
12	multiplicación	*
12	división	/
11	suma	+
11	resta	-
...	...	...
2	asignación	=
...	...	...

Asignación

Tener en cuenta que una asignación "=" también es un operador. Está listado en la tabla de precedencia con una prioridad muy baja de 2. Es por eso que, cuando asignamos una variable, como x = 2 * 2 + 1, los cálculos se realizan primero y luego se evalúa el =, almacenando el resultado en x.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

Asignación = devuelve un valor

Todos los operadores en JavaScript devuelven un valor. Esto es obvio para + y -, pero también es cierto para =. La llamada **x = value** escribe el value en x y luego lo devuelve.

Aquí hay una demostración que usa una asignación como parte de una expresión más compleja:

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

Asignaciones encadenadas

Otra característica interesante es la habilidad para encadenar asignaciones:

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Las asignaciones encadenadas evalúan de derecha a izquierda. Primero, se evalúa la expresión más a la derecha 2 + 2 y luego se asigna a las variables de la izquierda: c, b y a. Al final, todas las variables comparten un solo valor. Una vez más, con el propósito de la legibilidad es mejor separa tal código en unas pocas líneas.

```
c = 2 + 2;
b = c;
a = c;
```

Modificar en el lugar

A menudo necesitamos aplicar un operador a una variable y guardar el nuevo resultado en esa misma variable. Por ejemplo:

```
let n = 2;
n = n + 5;
n = n * 2;
```

```
let n = 2;
n += 5; // ahora n = 7  (es lo mismo que n = n + 5)
n *= 2; // ahora n = 14 (es lo mismo que n = n * 2)

alert( n ); // 14
```

* Esta notación puede ser acortada utilizando los operadores += y *=:

Los operadores cortos “modifica y asigna” existen para todos los operadores aritméticos y de nivel bit: /=, -=, etcétera. Tales operadores tienen la misma precedencia que la asignación normal, por lo tanto se ejecutan después de otros cálculos:

```
let n = 2;

n *= 3 + 5; // el lado derecho es evaluado primero, es lo mismo que n *= 8

alert( n ); // 16
```

Incremento/Decremento

Aumentar o disminuir un número en uno es una de las operaciones numéricas más comunes. Entonces, hay operadores especiales para ello:

- Incremento ++ incrementa una variable por 1:

```
1  let counter = 2;
2  counter++;      // funciona igual que counter = counter + 1, pero es más corto
3  alert( counter ); // 3
```

- Decremento -- decrementa una variable por 1:

```
1  let counter = 2;
2  counter--;      // funciona igual que counter = counter - 1, pero es más corto
3  alert( counter ); // 1
```

Los operadores ++ y -- pueden ser colocados antes o después de una variable.

- Cuando el operador va después de la variable, está en “forma de sufijo”: counter++.
- La “forma de prefijo” es cuando el operador va antes de la variable: ++counter.

Ambas sentencias hacen la misma cosa: aumentar counter por 1.

Todos los operadores devuelven un valor, Incremento/decremento no es una excepción. La forma prefijo devuelve el nuevo valor mientras que la forma sufijo devuelve el valor anterior (antes del incremento/decremento). Para ver la diferencia, aquí hay un ejemplo:

```
1  let counter = 1;
2  let a = ++counter; // (*)
3
4  alert(a); // 2
```

En la línea (*), la forma prefijo **++counter** incrementa *counter* y devuelve el nuevo valor,

2. Por lo tanto, el alert muestra 2. Ahora usemos la forma sufijo:

```
1  let counter = 1;
2  let a = counter++; // (*) cambiado ++counter a counter++
3
4  alert(a); // 1
```

En la línea (*), la forma sufijo **counter++** también incrementa *counter*, pero devuelve el antiguo valor (antes de incrementar). Por lo tanto, el alert muestra 1.

Para resumir:

- Si no se usa el resultado del incremento/decremento, no hay diferencia en la forma de usar:

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, las líneas de arriba realizan lo mismo
```

- Si queremos aumentar un valor y usar inmediatamente el resultado del operador, necesitamos la forma de prefijo:

```
let counter = 0;
alert( ++counter ); // 1
```

- Si queremos incrementar un valor, pero usamos su valor anterior, necesitamos la forma sufijo:

```
let counter = 0;
alert( counter++ ); // 0
```

Operadores a nivel de BIT

Los operadores a nivel bit tratan los argumentos como números enteros de 32 bits y trabajan en el nivel de su representación binaria. Estos operadores no son específicos de JavaScript. Son compatibles con la mayoría de los lenguajes de programación. La lista de operadores:

- AND (&)
- OR (|)
- XOR (^)
- NOT (~)
- LEFT SHIFT (<<)
- RIGHT SHIFT (>>)
- ZERO-FILL RIGHT SHIFT (>>>)

Coma

El operador coma "," es uno de los operadores más raros e inusuales. A veces, es utilizado para escribir código más corto. El operador coma nos permite evaluar varias expresiones, dividiéndolas con una coma ",". Cada una de ellas es evaluada, pero sólo el resultado de la última es devuelto. Por ejemplo:

```
1  let a = (1 + 2, 3 + 4);
2
3  alert( a ); // 7 (el resultado de 3 + 4)
```

Aquí, se evalúa la primera expresión 1 + 2 y se desecha su resultado. Luego, se evalúa 3 + 4 y se devuelve como resultado.

NOTA: Tenga en cuenta que el operador coma tiene una precedencia muy baja, inferior a =, por lo que los paréntesis son importantes en el ejemplo anterior. Sin ellos: a = 1 + 2, 3 + 4 se evalúa primero el +, sumando los números "a" a = 3, 7, luego el operador de asignación "=" asigna a = 3, y el resto es ignorado. Es igual que (a = 1 + 2), 3 + 4.

TAREAS

Las formas sufijo y prefijo

¿Cuáles son los valores finales de todas las variables `a`, `b`, `c` y `d` después del código a continuación?

```
1       let a = 1, b = 1;
2       let c = ++a; // ?
3       let d = b++; // ?
```

Resultado de asignación

¿Cuáles son los valores de 'a' y 'x' después del código a continuación?

```
1       let a = 2;
2     let x = 1 + (a *= 2);
```

Conversiones de tipos

¿Cuáles son los resultados de estas expresiones?

```
1       "" + 1 + 0
2       "" - 1 + 0
3       true + false
4       6 / "3"
5       "2" * "3"
6       4 + 5 + "px"
7       "$" + 4 + 5
8       "4" - 2
9       "4px" - 2
10      "  -9  " + 5
11      "  -9  " - 5
12      null + 1
13      undefined + 1
14      " \t \n" - 2
```

Piensa bien, anótalos y luego compara con la respuesta.

Corregir la adición

Aquí hay un código que le pide al usuario dos números y muestra su suma.

Funciona incorrectamente. El resultado en el ejemplo a continuación es 12 (para valores de captura predeterminados).

¿Por qué? Arréglalo. El resultado debería ser 3.

```
1       let a = prompt("¿Primer número?", 1);
2       let b = prompt("¿Segundo número?", 2);
3     alert(a + b); // 12
```

Comparaciones

En Javascript se escriben así:

- Mayor/menor que: a > b, a < b.
- Mayor/menor o igual que: a >= b, a <= b.
- Igual: a == b (ten en cuenta que el doble signo == significa comparación, mientras que un solo símbolo a = b significaría una asignación).
- Distinto. En matemáticas la notación es ≠, pero en JavaScript se escribe como una asignación con un signo de exclamación delante: a != b.

Booleano es el resultado

Como todos los demás operadores, una comparación retorna un valor. En este caso, el valor es un booleano.

- true – significa “sí”, “correcto” o “verdad”.
- false – significa “no”, “equivocado” o " no verdad".

Por ejemplo:

```
1  alert( 2 > 1 ); // true (correcto)
2  alert( 2 == 1 ); // false (incorrecto)
3  alert( 2 != 1 ); // true (correcto)
```

El resultado de una comparación puede asignarse a una variable, igual que cualquier valor:

```
1  let result = 5 > 4; // asignar el resultado de la comparación
2  alert( result ); // true
```

Comparación de cadenas

Para ver si una cadena es “mayor” que otra, JavaScript utiliza el llamado orden “de diccionario” o “lexicográfico”. En otras palabras, las cadenas se comparan letra por letra. Por ejemplo:

```
1  alert( 'Z' > 'A' ); // true
2  alert( 'Glow' > 'Glee' ); // true
3  alert( 'Bee' > 'Be' ); // true
```

El algoritmo para comparar dos cadenas es simple:

1) Compare el primer carácter de ambas cadenas.
2) Si el primer carácter de la primera cadena es mayor (o menor) que el de la otra cadena, entonces la primera cadena es mayor (o menor) que la segunda. Hemos terminado.
3) De lo contrario, si los primeros caracteres de ambas cadenas son los mismos, compare los segundos caracteres de la misma manera.
4) Repita hasta el final de cada cadena.
5) Si ambas cadenas tienen la misma longitud, entonces son iguales. De lo contrario, la cadena más larga es mayor.

En los ejemplos anteriores, la comparación 'Z' > 'A' llega a un resultado en el primer paso. La segunda comparación "Glow" y "Glee" necesitan más pasos, se comparan carácter por carácter:

1) G es igual que G.
2) l es igual que l.
3) es mayor que e. Detente aquí. La primera cadena es mayor.

NOTA: Por ejemplo, las mayúsculas importan. Una letra mayúscula "A" no es igual a la minúscula "a". ¿Cuál es mayor? La "a" minúscula. ¿Por qué? Porque el carácter en minúsculas tiene un mayor índice en la tabla de codificación interna que utiliza JavaScript (Unicode).

Comparación de diferentes tipos

Al comparar valores de diferentes tipos, JavaScript convierte los valores a números. Por ejemplo:

```
alert( '2' > 1 ); // true, la cadena '2' se convierte en el número 2
alert( '01' == 1 ); // true, la cadena '01' se convierte en el número 1
```

Para valores booleanos, true se convierte en 1 y false en 0. Por ejemplo:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Igualdad estricta

Una comparación regular de igualdad == tiene un problema. No puede diferenciar 0 de `falso':

```
alert( 0 == false ); // true
```

Lo mismo sucede con una cadena vacía:

```
alert( '' == false ); // true
```

Esto sucede porque los operandos de diferentes tipos son convertidos a números por el operador de igualdad "==". Una cadena vacía, al igual que false, se convierte en un cero.

¿Qué hacer si queremos diferenciar 0 de false? Un operador de igualdad estricto "===" comprueba la igualdad sin conversión de tipo. En otras palabras, si a y b son de diferentes tipos, entonces a === b retorna inmediatamente false sin intentar convertirlos. Intentémoslo:

```
alert( 0 === false ); // falso, porque los tipos son diferentes
```

Existe también un operador de diferencia estricta "!==" análogo a "!=".

Comparación con nulos e indefinidos

Hay un comportamiento no intuitivo cuando se compara null o undefined con otros valores. Para un control de igualdad estricto "===". Estos valores son diferentes, porque cada uno de ellos es de un tipo diferente.

```
1  alert( null === undefined ); // false
```

Para una comparación no estricta ==

Hay una regla especial. Estos dos son una " pareja dulce ": son iguales entre sí (en el sentido de ==), pero no a ningún otro valor.

```
1  alert( null == undefined ); // true
```

Resultados extraños: NULL vs 0

Comparemos null con un cero:

```
1  alert( null > 0 ); /// (1) false
2  alert( null == 0 ); /// (2) false
3  alert( null >= 0 ); // (3) true
```

La razón es que una comparación de igualdad == y las comparaciones > < >= <= funcionan de manera diferente. Las comparaciones convierten a null en un número, tratándolo como 0. Es por eso que (3) null >= 0 es verdadero y (1) null > 0 es falso.

Por otro lado, el control de igualdad == para undefined y null se define de tal manera que, sin ninguna conversión, son iguales entre sí y no son iguales a nada más. Es por eso que (2) null == 0 es falso.

Un indefinido incomparable

El valor undefined no debe compararse con otros valores:

```
1  alert( undefined > 0 ); // false (1)
2  alert( undefined < 0 ); // false (2)
3  alert( undefined == 0 ); // false (3)
```

Obtenemos estos resultados porque:

- Las comparaciones (1) y (2) retornan falso porque no definido se convierte en NaN y NaN es un valor numérico especial que retorna falso para todas las comparaciones.
- La comparación de igualdad (3) retorna falso porque undefined sólo equivale a null y a ningún otro valor.

RESUMEN

- Los operadores de comparación retornan un valor booleano.
- Las cadenas se comparan letra por letra en el orden del “diccionario”.
- Cuando se comparan valores de diferentes tipos, se convierten en números (excepto un control de igualdad estricta).
- Los valores null y undefined son iguales == entre sí y no equivalen a ningún otro valor.
- Ten cuidado al usar comparaciones como > o < con variables que ocasionalmente pueden ser null/undefined. Revisar por separado si hay null/undefined es una buena idea.

TAREAS

Comparaciones

¿Cuál será el resultado de las siguientes expresiones?

```
5 > 4
"apple" > "pineapple"
"2" > "12"
undefined == null
undefined === null
null == "\n0\n"
null === +"\n0\n"
```

EJECUCION CONDICIONAL

A veces necesitamos que, bajo condiciones diferentes, se ejecuten acciones diferentes. Para esto podemos usar la sentencia if y el "operador condicional" ?.

Sentencia "if"

La sentencia if(...) evalúa la condición en los paréntesis, y si el resultado es verdadero (true), ejecuta un bloque de código. Por ejemplo:

```
1  let year = prompt('¿En que año fué publicada la especificación ECMAScript-2015?');
2
3  if (year == 2015) alert( '¡Estás en lo cierto!' );
```

Aquí la condición es una simple igualdad (year == 2015), pero podría ser mucho más compleja. Si queremos ejecutar más de una sentencia, debemos encerrar nuestro bloque de código entre llaves:

```
1  if (year == 2015) {
2    alert( "¡Es Correcto!" );
3    alert( "¡Eres muy inteligente!" );
4  }
```

Recomendamos encerrar nuestro bloque de código entre llaves {} siempre que se utilice la sentencia if, incluso si solo se va a ejecutar una sola sentencia. Al hacerlo mejoramos la legibilidad.

Conversión Booleana

La sentencia if (…) evalúa la expresión dentro de sus paréntesis y convierte el resultado en booleano. Recordemos las reglas de conversión del capítulo Conversiones de Tipos:

- El número 0, un string vacío "", null, undefined, y NaN, se convierten en false. Por esto son llamados valores “falsos”.
- El resto de los valores se convierten en true, entonces los llamaremos valores “verdaderos”.

Se puede pasar un valor booleano pre-evaluado al if, así:

```
1  let cond = (year == 2015); // la igualdad evalúa y devuelve un true o false
2
3  if (cond) {
4    ...
5  }
```

La cláusula "else"

La sentencia if puede contener un bloque else ("si no", "en caso contrario") opcional. Este bloque se ejecutará cuando la condición sea falsa. Por ejemplo:

```
let year = prompt('¿En qué año fue publicada la especificación ECMAScript-2015?');

if (year == 2015) {
  alert( '¡Lo adivinaste, correcto!' );
} else {
  alert( '¿Cómo puedes estar tan equivocado?' ); // cualquier valor excepto 2015
}
```

Muchas condiciones "else if"

A veces se quiere comprobar más de una condición. La cláusula else if nos permite anidar varios condicionantes. Por ejemplo:

```
let year = prompt('¿En qué año fue publicada la especificación ECMAScript-2015?');

if (year < 2015) {
  alert( 'Muy poco...' );
} else if (year > 2015) {
  alert( 'Muy Tarde' );
} else {
  alert( '¡Exactamente!' );
}
```

En el código de arriba, JavaScript primero revisa si year < 2015. Si esto es falso, continúa a la siguiente condición year > 2015. Si esta también es falsa, mostrará la última alert. Podría haber más bloques else if. Y el último else es opcional.

Operador ternario "?"

A veces necesitamos que el valor que asignemos a una variable dependa de alguna condición. Por ejemplo:

```
let accessAllowed;
let age = prompt('¿Qué edad tienes?', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}

alert(accessAllowed);
```

El "operador condicional" nos permite ejecutar esto en una forma más corta y simple. El operador está representado por el signo de cierre de interrogación "?". A veces es llamado "ternario" porque el operador tiene tres operandos, es el único operador de JavaScript que tiene esa cantidad. La Sintaxis es:

```
let result = condition ? value1 : value2;
```

Se evalúa condition: si es verdadera entonces devuelve value1 , de lo contrario value2. Por ejemplo:

```
let accessAllowed = (age > 18) ? true : false;
```

Técnicamente, podemos omitir el paréntesis alrededor de age > 18. El operador de signo de interrogación tiene una precedencia baja, por lo que se ejecuta después de la comparación >. En este ejemplo realizaremos lo mismo que en el anterior:

```
// el operador de comparación  "age > 18" se ejecuta primero de cualquier forma
// (no necesitamos agregar los paréntesis)
let accessAllowed = age > 18 ? true : false;
```

NOTA: En el ejemplo de arriba, podrías evitar utilizar el operador de signo de interrogación porque esta comparación devuelve directamente true/false.

Múltiples

Una secuencia de operadores de signos de interrogación “?” puede devolver un valor que depende de más de una condición. Por ejemplo:

```
let age = prompt('¿edad?', 18);

let message = (age < 3) ? '¡Hola, bebé!' :
  (age < 18) ? '¡Hola!' :
  (age < 100) ? '¡Felicidades!' :
  '¡Qué edad tan inusual!';

alert( message );
```

Puede ser difícil al principio comprender lo que está sucediendo. Pero después de una mirada más cercana, podemos ver que es solo una secuencia ordinaria de condiciones:

1) El primer signo de pregunta revisa si age < 3.
2) Si es cierto, devuelve '¡Hola, bebé!'. De lo contrario, continúa a la expresión que está después de los dos puntos “:”, la cual revisa si age < 18.
3) Si es cierto, devuelve '¡Hola!'. De lo contrario, continúa con la expresión que está después de los dos puntos siguientes “:”, la cual revisa si age < 100.
4) Si es cierto, devuelve '¡Felicidades!'. De lo contrario, continúa a la expresión que está después de los dos puntos “:”, la cual devuelve '¡Qué edad tan inusual!'.

Aquí lo podemos ver utilizando if..else:

```
if (age < 3) {
  message = '¡Hola, bebé!';
} else if (age < 18) {
  message = '¡Hola!';
} else if (age < 100) {
  message = '¡Felicidades!';
} else {
  message = '¡Qué edad tan inusual!';
}
```

A veces, el signo de interrogación de cierre “?” se utiliza para reemplazar un if:

```
let company = prompt('¿Qué compañía creó JavaScript?', '');

(company == 'Netscape') ?
   alert('¡Correcto!') : alert('Equivocado.');
```

Dependiendo de la condición company == 'Netscape', se ejecutará la primera o la segunda expresión del operador “?” y se mostrará una alerta. Aquí no asignamos el resultado a una variable. En vez de esto, ejecutamos diferentes códigos dependiendo de la condición.

La notación es más corta que la sentencia equivalente con “if”, lo cual seduce a algunos programadores. Pero es menos legible. Aquí está el mismo código utilizando la sentencia if para comparar:

```
let company = prompt('¿Cuál compañía creó JavaScript?', '');

if (company == 'Netscape') {
  alert('¡Correcto!');
} else {
  alert('Equivocado.');
}
```

Los bloques de código que se expanden múltiples líneas son más fáciles de entender que los las instrucciones largas horizontales. El propósito del operador de signo de interrogación “?” es para devolver un valor u otro dependiendo de su condición. Por favor utilízala para exactamente esto. Utiliza la sentencia “if” cuando necesites ejecutar código en ramas distintas.

TAREAS

if (un string con cero)

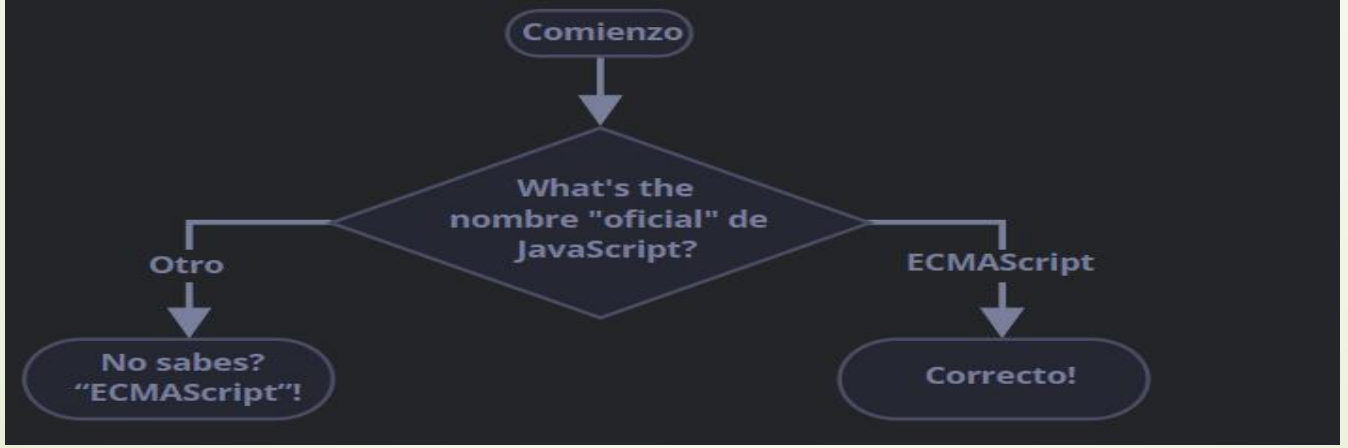
Se mostrará el `alert`?

```
if ("0") {
  alert( 'Hello' );
}
```

El nombre de JavaScript

Usando el constructor **`if..else`**, escribe el código que pregunta: '¿Cuál es el nombre "oficial" de JavaScript?'

Si el visitante escribe "ECMAScript", entonces muestra: "¡Correcto!", de lo contrario muestra: "¿No lo sabes? ¡ECMAScript!"



Demo en nueva ventana

Muestra el signo

Usando el constructor `if..else`, escribe un código que obtenga a través de un `prompt` un número y entonces muestre en un `alert`:

- `1`, si el valor es mayor que cero,
- `-1`, si es menor que cero,
- `0`, si es igual a cero.

En la tarea asumimos que siempre el usuario introduce un número.

Demo en nueva ventana

Reescriba el 'if..else' con '?'

Reescriba el `if..else` utilizando operadores ternarios múltiples`'?'`.

Para legibilidad, es recomendad dividirlo en múltiples lineas de código.

```
let message;

if (login == 'Empleado') {
  message = 'Hola';
} else if (login == 'Director') {
  message = 'Felicidades';
} else if (login == '') {
  message = 'Sin sesión';
} else { message = '';}
```

Operadores Lógicos

Hay cuatro operadores lógicos en JavaScript: || (O), && (Y), ! (NO), ?? (Fusión de nulos). Aunque sean llamados lógicos, pueden ser aplicados a valores de cualquier tipo, no solo booleanos. El resultado también puede ser de cualquier tipo.

|| (OR)

El operador OR se representa con dos símbolos de linea vertical:

```
result = a || b;
```

En la programación clásica, el OR lógico está pensado para manipular solo valores booleanos. Si cualquiera de sus argumentos es true, retorna true, de lo contrario retorna false. En JavaScript, el operador es un poco más complicado y poderoso. Pero primero, veamos qué pasa con los valores booleanos. Hay cuatro combinaciones lógicas posibles:

```
alert(true || true); // true (verdadero)
alert(false || true); // true
alert(true || false); // true
alert(false || false); // false (falso)
```

Como podemos ver, el resultado es siempre true excepto cuando ambos operandos son false. Si un operando no es un booleano, se lo convierte a booleano para la evaluación. Por ejemplo, el número 1 es tratado como true, el número 0 como false:

```
if (1 || 0) { // Funciona como if( true || false )
  alert("valor verdadero!");
}
```

La mayoría de las veces, OR || es usado en una declaración if para probar si alguna de las condiciones dadas es true. Por ejemplo:

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'La oficina esta cerrada.' );
}
```

Podemos pasar más condiciones:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert("La oficina esta cerrada."); // Es fin de semana
}
```

|| (OR) encuentra el primer valor verdadero

La lógica descrita arriba es algo clásica. Ahora, mostremos las características “extra” de JavaScript. El algoritmo extendido trabaja de la siguiente forma. Dado múltiples valores aplicados al operador OR:

```
result = value1 || value2 || value3;
```

El operador OR || realiza lo siguiente:

- Evalúa los operandos de izquierda a derecha.
- Para cada operando, convierte el valor a booleano. Si el resultado es true, se detiene y retorna el valor original de ese operando.
- Si todos los operandos han sido evaluados (todos eran false), retorna el ultimo operando.

Un valor es retornado en su forma original, sin la conversión. En otras palabras, una cadena de OR "||" devuelve el primer valor verdadero o el último si ningún verdadero es encontrado. Por ejemplo:

```
alert(1 || 0); // 1 (1 es un valor verdadero)

alert(null || 1); // 1 (1 es el primer valor verdadero)
alert(null || 0 || 1); // 1 (el primer valor verdadero)

alert(undefined || null || 0); // 0 (todos son valores falsos, retorna el último valor)
```

Esto brinda varios usos interesantes comparados al “OR puro, clásico, de solo booleanos”.

1) Obtener el primer valor verdadero de una lista de variables o expresiones. Por ejemplo, tenemos las variables firstName, lastName y nickName, todas opcionales (pueden ser undefined o tener valores falsos).
Usemos OR || para elegir el que tiene los datos y mostrarlo (o anónimo si no hay nada configurado):

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

Si todas las variables fueran falsas, aparecería "Anonymous".

2) Evaluación del camino más corto.
Otra característica del operador OR || es la evaluación de “el camino más corto” o “cortocircuito”. Esto significa que || procesa sus argumentos hasta que se alcanza el primer valor verdadero, y ese valor se devuelve inmediatamente sin siquiera tocar el otro argumento.
La importancia de esta característica se vuelve obvia si un operando no es solo un valor sino una expresión con un efecto secundario, como una asignación de variable o una llamada a función.
En el siguiente ejemplo, solo se imprime el segundo mensaje:

```
true || alert("not printed");
false || alert("printed");
```

En la primera línea, el operador OR || detiene la evaluación inmediatamente después de ver que es verdadera, por lo que la alerta no se ejecuta. A veces se usa esta función para ejecutar comandos solo si la condición en la parte izquierda es falsa.

&& (AND)

El operador AND es representado con dos ampersands &&:

```
result = a && b;
```

En la programación clásica, AND retorna true si ambos operandos son valores verdaderos y false en cualquier otro caso.

```
alert(true && true); // true
alert(false && true); // false
alert(true && false); // false
alert(false && false); // false
```

Un ejemplo con if:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert("La hora es 12:30");
}
```

Al igual que con OR, cualquier valor es permitido como operando de AND:

```
if (1 && 0) { // evaluado como true && false
  alert( "no funcionará porque el resultado es un valor falso" );
}
```

AND “&&” encuentra el primer valor falso

Dado múltiples valores aplicados al operador AND:

```
result = value1 && value2 && value3;
```

El operador AND && realiza lo siguiente:

- Evalúa los operandos de izquierda a derecha.
- Para cada operando, los convierte a un booleano. Si el resultado es false, se detiene y retorna el valor original de dicho operando.
- Si todos los operandos han sido evaluados (todos fueron valores verdaderos), retorna el último operando.

En otras palabras, AND retorna el primer valor falso o el último valor si ninguno fue encontrado. Las reglas anteriores son similares a las de OR. La diferencia es que AND retorna el primer valor falso mientras que OR retorna el primer valor verdadero. Ejemplo:

```
// si el primer operando es un valor verdadero,
// AND retorna el segundo operando:
alert(1 && 0); // 0
alert(1 && 5); // 5

// si el primer operando es un valor falso,
// AND lo retorna. El segundo operando es ignorado
alert(null && 5); // null
alert(0 && "cualquier valor"); // 0
```

También podemos pasar varios valores de una vez. Observa como el primer valor falso es retornado:

```
1  alert(1 && 2 && null && 3); // null
```

Cuando todos los valores son verdaderos, el último valor es retornado:

```
1  alert(1 && 2 && 3); // 3, el último.
```

NOTA: La precedencia del operador AND && es mayor que la de OR ||. Así que el código a && b || c && d es básicamente el mismo que si las expresiones && estuvieran entre paréntesis: (a && b) || (c && d)

! (NOT)

El operador booleano NOT se representa con un signo de exclamación "!". La sintaxis es bastante simple:

```
result = !value;
```

El operador acepta un solo argumento y realiza lo siguiente:

- Convierte el operando al tipo booleano: true/false.
- Retorna el valor contrario.

Por ejemplo:

```
alert(!true); // false
alert(!0); // true
```

Un doble NOT !! es a veces usado para convertir un valor al tipo booleano:

```
alert(!!"cadena de texto no vacía"); // true
alert(!!null); // false
```

Eso es, el primer NOT convierte el valor a booleano y retorna el inverso, y el segundo NOT lo invierte de nuevo. Al final, tenemos una simple conversión a booleano. Hay una manera un poco más prolija de realizar lo mismo – una función integrada Boolean:

```
alert(Boolean("cadena de texto no vacía")); // true
alert(Boolean(null)); // false
```

NOTA: La precedencia de NOT ! es la mayor de todos los operadores lógicos, así que siempre se ejecuta primero, antes que && o ||.

TAREAS

¿Cuál es el resultado de OR?

¿Cuál será la salida del siguiente código?

```
alert( null || 2 || undefined );
```

¿Cuál es el resultado de las alertas aplicadas al operador OR?

¿Cuál será la salida del siguiente código?

```
alert( alert(1) || 2 || alert(3) );
```

¿Cuál es el resultado de AND?

¿Cuál será la salida del siguiente código?

```
alert( 1 && null && 2 );
```

¿Cuál es el resultado de las alertas aplicadas al operador AND?

¿Cuál será la salida del siguiente código?

```
alert( alert(1) && alert(2) );
```

El resultado de OR AND OR

¿Cuál será el resultado? `alert( null || 2 && 3 || 4 );`

Comprueba el rango por dentro

Escribe una condición "if" para comprobar que `age`(edad) está entre `14` y `90` inclusive.

"Inclusive" significa que `age` puede llegar a ser uno de los extremos, `14` o `90`.

Comprueba el rango por fuera

Escribe una condición `if` para comprobar que `age` NO está entre 14 y 90 inclusive.

Crea dos variantes: la primera usando NOT `!`, y la segunda sin usarlo.

Una pregunta acerca de "if"

¿Cuáles de estos `alert`s va a ejecutarse?

¿Cuáles serán los resultados de las expresiones dentro de `if(...)`?

```
if (-1 || 0) alert( "primero" );
if (-1 && 0) alert( "segundo" );
if (null || -1 && 1) alert( "tercero" );
```

Comprueba el inicio de sesión

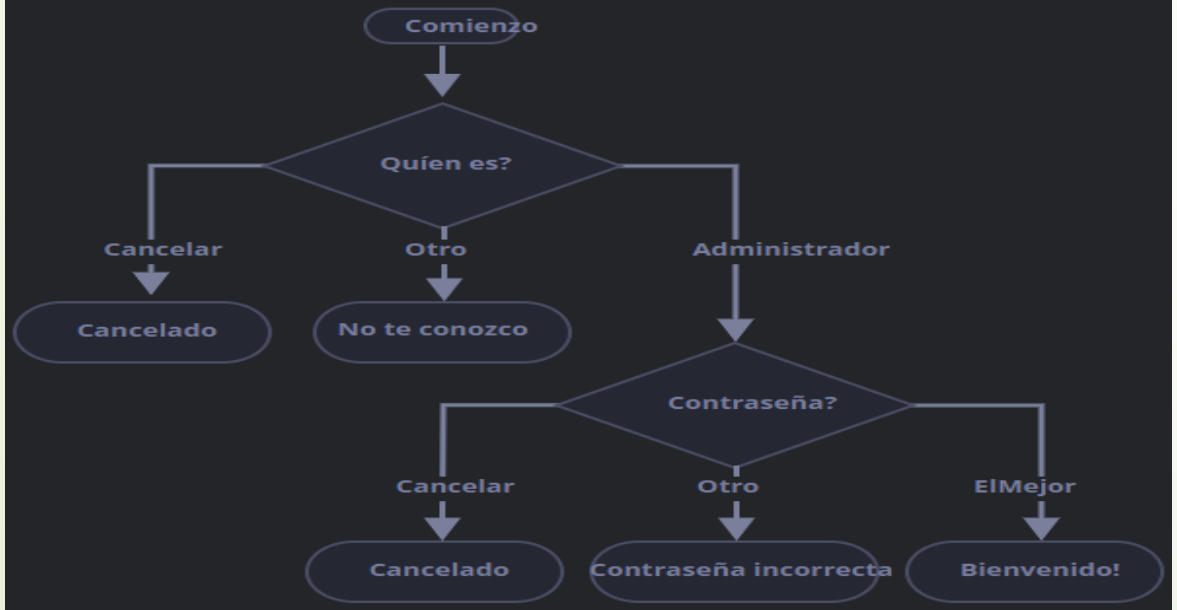
Escribe un código que pregunte por el inicio de sesión con `propmt`.

Si el visitante ingresa `"Admin"`, entonces `prompt`(pregunta) por una contraseña, si la entrada es una linea vacía o Esc – muestra "Cancelado.", si es otra cadena de texto – entonces muestra "No te conozco".

La contraseña se comprueba de la siguiente manera:

- Si es igual a "TheMaster", entonces muestra "Bienvenido!",
- Si es otra cadena de texto – muetra "Contraseña incorrecta",
- Para una cadena de texto vacía o una entrada cancelada, muestra "Cancelado."

El esquema:



Por favor usa bloques anidados de `if`. Piensa en la legibilidad general del código.

Ejecutar el demo

Operador Nullish Coalescing '??'

NOTA: Esta es una adición reciente al lenguaje. Los navegadores antiguos pueden necesitar polyfills.

El operador “nullish coalescing” (fusión de null) se escribe con un doble signo de cierre de interrogación “??”. Como esta trata a null y a undefined de forma similar, usaremos un término especial para este artículo. Diremos que una expresión es “definida” cuando no es null ni undefined. El resultado de a ?? b:

- si a está “definida”, será a,
- si a no está “definida”, será b.

Es decir, “??” devuelve el primer argumento cuando este no es null ni undefined. En caso contrario, devuelve el segundo. El operador “nullish coalescing” no es algo completamente nuevo. Es solamente una sintaxis agradable para obtener el primer valor “definido” de entre dos. Podemos reescribir result = a ?? b usando los operadores que ya conocemos:

```
result = (a !== null && a !== undefined) ? a : b;
```

Ahora debería estar absolutamente claro lo que ?? hace. Veamos dónde podemos utilizarlo. El uso típico de ?? es brindar un valor predeterminado. Por ejemplo, aquí mostramos user si su valor está “definido” (que no es null ni undefined). De otro modo, muestra “Anonymous”:

```
let user;

alert(user ?? "Anonymous"); // Anonymous (user es undefined)
```

Aquí el ejemplo de “user” con un nombre asignado:

```
let user = "John";

alert(user ?? "Anonymous"); // John (user no es null ni undefined)
```

También podemos usar una secuencia de ?? para seleccionar el primer valor que no sea null/undefined de una lista. Digamos que tenemos los datos de un usuario en las variables firstName, lastName y nickName. Todos ellos podrían ser indefinidos si el usuario decide no ingresar los valores correspondientes.

Queremos mostrar un nombre usando una de estas variables, o mostrar “anónimo” si todas ellas son null/undefined. Usemos el operador ?? para ello:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// Muestra el primer valor definido:
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder
```

Comparación con ||

El operador OR || puede ser usado de la misma manera que ??, tal como está explicado en el capítulo previo. Por ejemplo, en el código de arriba podemos reemplazar ?? por || y obtener el mismo resultado:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// muestra el primer valor "verdadero":
alert(firstName || lastName || nickName || "Anonymous"); // Supercoder
```

La gran diferencia es que:

- || devuelve el primer valor verdadero.
- ?? devuelve el primer valor definido.

El || no distingue entre false, 0, un string vacío "", y null/undefined. Todos son lo mismo: valores “falsos”. Si cualquiera de ellos es el primer argumento de ||, obtendremos el segundo argumento como resultado.

Pero en la práctica podemos querer usar el valor predeterminado solamente cuando la variable es null/undefined, es decir cuando el valor realmente es desconocido o no fue establecido. Por ejemplo considera esto:

```
let height = 0; // altura cero

alert(height || 100); // 100
alert(height ?? 100); // 0
```

height || 100 verifica si height es “falso”, y 0 lo es. - así el resultado de || es el segundo argumento, 100. height ?? 100 verifica si height es null/undefined, y no lo es. - así el resultado es height como está, que es 0.

Precedencia

La precedencia del operador ?? es la misma de ||. Ambos son iguales a 3 en la Tabla MDN. Esto significa que ambos operadores, || y ??, son evaluados antes que = y ?, pero después de la mayoría de las demás operaciones como + y *. Así que podemos necesitar añadir paréntesis:

```
let height = null;
let width = null;

// Importante: usar paréntesis
let area = (height ?? 100) * (width ?? 50);

alert(area); // 5000
```

Caso contrario, si omitimos los paréntesis, entonces * tiene una mayor precedencia y se ejecutará primero. Eso sería lo mismo que:

```
// sin paréntesis
let area = height ?? 100 * width ?? 50;

// ...funciona de esta forma (no es lo que queremos):
let area = height ?? (100 * width) ?? 50;
```

Uso de ?? con && y ||

Por motivos de seguridad, JavaScript prohíbe el uso de “??” junto con los operadores “&&” y “||”, salvo que la precedencia sea explícitamente especificada con paréntesis. El siguiente código desencadena un error de sintaxis:

```
1  let x = 1 && 2 ?? 3; // Syntax error
```

La limitación es debatible. Fue agregada a la especificación del lenguaje con propósito de evitar equivocaciones cuando la gente comenzara a reemplazar “||” por “??”. Usa paréntesis explícitos para solucionarlo:

```
1  let x = (1 && 2) ?? 3; // Funciona
2
3  alert(x); // 2
```

RESUMEN

- El operador “nullish coalescing” ?? brinda una manera concisa de seleccionar un valor “definido” de una lista.
- Es usado para asignar valores por defecto a las variables:

```
1  // Asignar height=100, si height es null o undefined
2  height = height ?? 100;
```

- El operador ?? tiene una precedencia muy baja, un poco más alta que ? y =.
- Está prohibido su uso con || y && sin paréntesis explícitos.