UD 4. JAVASCRIPT

Índex

1. INTRODUCCIÓN

1.1.¿Qué es JavaScript?

JavaScript fue creado para “dar vida a las páginas web”. Los programas en este lenguaje se llaman scripts. Se pueden escribir directamente en el HTML de una página web y ejecutarse automáticamente a medida que se carga la página.

Los scripts se proporcionan y ejecutan como texto plano. No necesitan preparación especial o compilación para correr. En este aspecto, JavaScript es muy diferente a otro lenguaje llamado Java.

1.2.¿Por qué se llama JavaScript?

Cuando JavaScript fue creado, inicialmente tenía otro nombre: “LiveScript”. Pero Java era muy popular en ese momento, así que se decidió que el posicionamiento de un nuevo lenguaje como un “Hermano menor” de Java ayudaría.

Pero a medida que evolucionaba, JavaScript se convirtió en un lenguaje completamente independiente con su propia especificación llamada ECMAScript, y ahora no tiene ninguna relación con Java.

Hoy, JavaScript puede ejecutarse no solo en los navegadores, sino también en servidores o incluso en cualquier dispositivo que cuente con un programa especial llamado El motor o intérprete de JavaScript.

El navegador tiene un motor embebido a veces llamado una “Máquina virtual de JavaScript”. Diferentes motores tienen diferentes “nombres en clave”. Por ejemplo:

- V8 – en Chrome, Opera y Edge.
- SpiderMonkey – en Firefox.
- Existen otros nombres en clave como “Chakra” para IE , “JavaScriptCore”, “Nitro” y “SquirrelFish” para Safari, etc.

Es bueno recordar estos términos porque son usados en artículos en internet.

1.3.¿Como trabajan los motores?

Los motores son complicados, pero los fundamentos son fáciles.

- El motor (embebido si es un navegador) lee (“analiza”) el script.
- Luego convierte (“compila”) el script a lenguaje de máquina.
- Por último, el código máquina se ejecuta, muy rápido.

El motor aplica optimizaciones en cada paso del proceso. Incluso observa como el script compilado se ejecuta, analiza los datos que fluyen a través de él y aplica optimizaciones al código maquina basadas en ese conocimiento.

1.4.¿Qué puede hacer JavaScript en el navegador?

El JavaScript moderno es un lenguaje de programación "seguro". No proporciona acceso de bajo nivel a la memoria ni a la CPU (UCP); ya que se creó inicialmente para los navegadores, los cuales no lo requieren.

Las capacidades de JavaScript dependen en gran medida en el entorno en que se ejecuta. Por ejemplo, Node.JS soporta funciones que permiten a JavaScript leer y escribir archivos arbitrariamente, realizar solicitudes de red, etc.

En el navegador JavaScript puede realizar cualquier cosa relacionada con la manipulación de una página web, interacción con el usuario y el servidor web.

Por ejemplo, en el navegador JavaScript es capaz de:

- Agregar nuevo HTML a la página, cambiar el contenido existente y modificar estilos.
- Reaccionar a las acciones del usuario, ejecutarse con los clics del ratón, movimientos del puntero y al oprimir teclas.
- Enviar solicitudes de red a servidores remotos, descargar y cargar archivos (Tecnologías llamadas AJAX y COMET).
- Obtener y configurar cookies, hacer preguntas al visitante y mostrar mensajes.
- Recordar datos en el lado del cliente con el almacenamiento local ("local storage").

1.5.¿Qué NO PUEDE hacer JavaScript en el navegador?

Las capacidades de JavaScript en el navegador están limitadas para proteger la seguridad de usuario. El objetivo es evitar que una página maliciosa acceda a información privada o dañe los datos de usuario.

Ejemplos de tales restricciones incluyen:

- JavaScript en el navegador no puede leer y escribir arbitrariamente archivos en el disco duro, copiarlos o ejecutar programas. No tiene acceso directo a funciones del Sistema operativo (OS).
- Los navegadores más modernos le permiten trabajar con archivos, pero el acceso es limitado y solo permitido si el usuario realiza ciertas acciones, como "arrastrar" un archivo a la ventana del navegador o seleccionarlo por medio de una etiqueta <input>.
- Existen maneras de interactuar con la cámara, micrófono y otros dispositivos, pero eso requiere el permiso explícito del usuario.

- Diferentes pestañas y ventanas generalmente no se conocen entre sí. A veces sí lo hacen: por ejemplo, cuando una ventana usa JavaScript para abrir otra. Pero incluso en este caso, JavaScript no puede acceder a la otra si provienen de diferentes sitios (de diferente dominio, protocolo o puerto).

  Esta restricción es conocida como "política del mismo origen" ("Same Origin Policy"). Es posible la comunicación, pero ambas páginas deben acordar el intercambio de datos y también deben contener el código especial de JavaScript que permite controlarlo.

- JavaScript puede fácilmente comunicarse a través de la red con el servidor de donde la página actual proviene. Pero su capacidad para recibir información de otros sitios y dominios está bloqueada. Aunque sea posible, esto requiere un acuerdo explícito (expresado en los encabezados HTTP) desde el sitio remoto. Una vez más: esto es una limitación de seguridad.

Tales limitaciones no existen si JavaScript es usado fuera del navegador; por ejemplo, en un servidor. Los navegadores modernos también permiten complementos y extensiones que pueden solicitar permisos extendidos.

1.6. ¿Qué hace a JavaScript único?

Existen al menos tres cosas geniales sobre JavaScript:

- Completa integración con HTML y CSS.
- Las cosas simples se hacen de manera simple.
- Soportado por la mayoría de los navegadores y habilitado de forma predeterminada.
- JavaScript es la única tecnología de los navegadores que combina estas tres cosas.

Eso es lo que hace a JavaScript único. Por esto es la herramienta mas extendida para crear interfaces de navegador.

Dicho esto, JavaScript también permite crear servidores, aplicaciones móviles, etc.

2. HOLA MUNDO

2.1. La etiqueta “script”.

Los programas de JavaScript se pueden insertar en casi cualquier parte de un documento HTML con el uso de la etiqueta <script>.

Por ejemplo:

```
<!DOCTYPE HTML>
<html>
    <body>
            <p>Antes del script...</p>
            <script>
                alert( '¡Hola, mundo!' );
            </script>
            <p>...Después del script.</p>
    </body>
</html>
```

Puedes ejecutar el ejemplo haciendo clic en el botón “Play” en la esquina superior derecha del cuadro de arriba.

La etiqueta <script> contiene código JavaScript que se ejecuta automáticamente cuando el navegador procesa la etiqueta.

2.2. Marcado Moderno

La etiqueta <script> tiene algunos atributos que rara vez se usan en la actualidad, pero aún se pueden encontrar en código antiguo:

El atributo type: <script type=…>

El antiguo estándar HTML, HTML4, requería que un script tuviera un type. Por lo general, era type="text/javascript". Ya no es necesario. Además, el estándar HTML moderno cambió totalmente el significado de este atributo. Ahora, se puede utilizar para módulos de JavaScript. Pero eso es un tema avanzado, hablaremos sobre módulos en otra parte del tutorial.

El atributo language: <script language=…>

Este atributo estaba destinado a mostrar el lenguaje del script. Este atributo ya no tiene sentido porque JavaScript es el lenguaje predeterminado. No hay necesidad de usarlo.

Scripts externos

Si tenemos un montón de código JavaScript, podemos ponerlo en un archivo separado.

Los archivos de script se adjuntan a HTML con el atributo src:

```
<script src="/path/to/script.js"></script>
```

Aquí, /path/to/script.js es una ruta absoluta al archivo de script desde la raíz del sitio. También se puede proporcionar una ruta relativa desde la página actual. Por ejemplo, src="script.js" significaría un archivo "script.js" en la carpeta actual.

También podemos dar una URL completa. Por ejemplo:

```
<script src="https://linkarecurso.com/script.js"></script>
```

Para adjuntar varios scripts, usa varias etiquetas:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
...
```

Como regla general, solo los scripts más simples se colocan en el HTML. Los más complejos residen en archivos separados.

La ventaja de un archivo separado es que el navegador lo descargará y lo almacenará en caché.

Otras páginas que hacen referencia al mismo script lo tomarán del caché en lugar de descargarlo, por lo que el archivo solo se descarga una vez.

Eso reduce el tráfico y hace que las páginas sean más rápidas.

Si se establece src, el contenido del script se ignora.

Una sola etiqueta <script> no puede tener el atributo src y código dentro.

3. CONTROL DE FLUJO

3.1.Sentencias

Las sentencias son construcciones sintácticas y comandos que realizan acciones.

Ya hemos visto una sentencia, alert('¡Hola mundo!'), que muestra el mensaje “¡Hola mundo!”.

Podemos tener tantas sentencias en nuestro código como queramos, las cuales se pueden separar con un punto y coma.

Por ejemplo, aquí separamos “Hello World” en dos alerts:

```
alert('Hola'); alert('Mundo');
```

Generalmente, las sentencias se escriben en líneas separadas para hacer que el código sea más legible:

```
alert('Hola');
alert('Mundo');
```

3.2.Punto y coma

Se puede omitir un punto y coma en la mayoría de los casos cuando existe un salto de línea.

Esto también funcionaría:

```
alert('Hola')
alert('Mundo')
```

Aquí, JavaScript interpreta el salto de línea como un punto y coma “implícito”. Esto se denomina inserción automática de punto y coma.

En la mayoría de los casos, una nueva línea implica un punto y coma. Pero “en la mayoría de los casos” no significa “siempre”!

Hay casos en que una nueva línea no significa un punto y coma. Por ejemplo:

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

El código da como resultado 6 porque JavaScript no inserta punto y coma aquí. Es intuitivamente obvio que si la línea termina con un signo más "+", es una “expresión incompleta”, un punto y coma aquí sería incorrecto. Y en este caso eso funciona según lo previsto.

Pero hay situaciones en las que JavaScript “falla” al asumir un punto y coma donde realmente se necesita.

Los errores que ocurren en tales casos son bastante difíciles de encontrar y corregir.

Recomendamos colocar puntos y coma entre las sentencias, incluso si están separadas por saltos de línea. Esta regla está ampliamente adoptada por la comunidad. Notemos una vez más que es posible omitir los puntos y coma la mayoría del tiempo. Pero es más seguro, especialmente para un principiante, usarlos.

3.3.Comentarios

A medida que pasa el tiempo, los programas se vuelven cada vez más complejos. Se hace necesario agregar comentarios que describan lo que hace el código y por qué.

Los comentarios se pueden poner en cualquier lugar de un script. No afectan su ejecución porque el motor simplemente los ignora.

Los comentarios de una línea comienzan con dos caracteres de barra diagonal //.

El resto de la línea es un comentario. Puede ocupar una línea completa propia o seguir una sentencia.

Como aquí:

```
// Este comentario ocupa una línea propia.
alert('Hello');

alert('World'); // Este comentario sigue a la sentencia.
```

Los comentarios de varias líneas comienzan con una barra inclinada y un asterisco /* y terminan con un asterisco y una barra inclinada */.

Como aquí:

```
/* Un ejemplo con dos mensajes.
Este es un comentario multilínea.
*/
alert('Hola');
alert('Mundo');
El contenido de los comentarios se ignora, por lo que si colocamos el código dentro de /* … */, no se ejecutará.
```

¡Los comentarios anidados no son admitidos! No puede haber /*...*/ dentro de otro /*...*/.

Los comentarios aumentan el tamaño general del código, pero eso no es un problema en absoluto. Hay muchas herramientas que minimizan el código antes de publicarlo en un servidor de producción. Eliminan los comentarios, por lo que no aparecen en los scripts de trabajo. Por lo tanto, los comentarios no tienen ningún efecto negativo en la producción.

4. EL MODO MODERNO “USE STRICT”

Durante mucho tiempo, JavaScript evolucionó sin problemas de compatibilidad. Se añadían nuevas características al lenguaje sin que la funcionalidad existente cambiase.

Esto tenía el beneficio de nunca romper código existente, pero lo malo era que cualquier error o decisión incorrecta tomada por los creadores de JavaScript se quedaba para siempre en el lenguaje.

Esto fue así hasta 2009, cuando ECMAScript 5 (ES5) apareció. Esta versión añadió nuevas características al lenguaje y modificó algunas de las ya existentes. Para mantener el código antiguo funcionando, la mayor parte de las modificaciones están desactivadas por defecto. Tienes que activarlas explícitamente usando una directiva especial: "use strict".

4.1. “use strict”

La directiva se asemeja a un string: "use strict". Cuando se sitúa al principio de un script, el script entero funciona de la manera “moderna”.

Por ejemplo:

```
"use strict";
// este código funciona de la manera moderna
...
```

Aprenderemos funciones (una manera de agrupar comandos) en breve, pero adelantemos que "use strict" se puede poner al inicio de una función. De esta manera, se activa el modo estricto únicamente en esa función. Pero normalmente se utiliza para el script entero.

Asegúrate de que “use strict” está al inicio

Por favor, asegúrate de que "use strict" está al principio de tus scripts. Si no, el modo estricto podría no estar activado.

El modo estricto no está activado aquí:

```
alert("algo de código");
// la directiva "use strict" de abajo es ignorada, tiene que estar al principio
"use strict";
// el modo estricto no está activado
```

Únicamente pueden aparecer comentarios por encima de "use strict".

4.2. Consola del navegador

Cuando utilices la consola del navegador para ejecutar código, ten en cuenta que no utiliza use strict por defecto.

En ocasiones, donde use strict cause diferencia, obtendrás resultados incorrectos. Entonces, ¿como utilizar use strict en la consola?

Primero puedes intentar pulsando Shift+Enter para ingresar múltiples líneas y poner use strict al principio, como aquí:

```
'use strict'; <Shift+Enter para una nueva línea>
//  ...tu código
<Intro para ejecutar>
```

Esto funciona para la mayoría de los navegadores, específicamente Firefox y Chrome.

Si esto no funciona, como en los viejos navegadores, hay una fea pero confiable manera de asegurar use strict. Ponlo dentro de esta especie de envoltura:

```
(function() {
  'use strict';


  // ...tu código...
})()
```

4.3. ¿Deberíamos utilizar “use strict”?

La pregunta podría parecer obvia, pero no lo es.

Uno podría recomendar que se comiencen los script con "use strict"… ¿Pero sabes lo que es interesante?

El JavaScript moderno admite “clases” y “módulos”, estructuras de lenguaje avanzadas, que automáticamente habilitan use strict. Entonces no necesitamos agregar la directiva "use strict" si las usamos.

Entonces, por ahora "use strict"; es un invitado bienvenido al tope de tus scripts. Luego, cuando tu código sea todo clases y módulos, puedes omitirlo.

5. VARIABLES Y TIPOS DE DATOS

La mayoría del tiempo, una aplicación de JavaScript necesita trabajar con información. Aquí hay 2 ejemplos:

- Una tienda en línea – La información puede incluir los bienes a la venta y un "carrito de compras".
- Una aplicación de chat – La información puede incluir los usuarios, mensajes, y mucho más.

Utilizamos las variables para almacenar esta información.

5.1. Una Variable

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) una variable con el nombre "message":

```
let message;
```

Ahora podemos introducir datos en ella al utilizar el operador de asignación =:

```
let message;
message = 'Hola'; // almacenar la cadena 'Hola' en la variable llamada message
```

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

```
let message;
message = 'Hola!';


alert(message); // muestra el contenido de la variable
```

Para ser concisos, podemos combinar la declaración de la variable y su asignación en una sola línea:

```
 let message = 'Hola!'; // define la variable y asigna un valor


alert(message); // Hola!
```

5.2.var en vez de let

En scripts más viejos, a veces se encuentra otra palabra clave: var en lugar de let:

```
var mensaje = 'Hola';
```

La palabra clave var es casi lo mismo que let. También hace la declaración de una variable, aunque de un modo ligeramente distinto, y más antiguo.

Existen sutiles diferencias entre let y var, pero no nos interesan en este momento.

5.3.Nombramiento de variables

Existen dos limitaciones de nombre de variables en JavaScript:

- El nombre únicamente puede incluir letras, dígitos, o los símbolos $ y _.
- El primer carácter no puede ser un dígito.

Ejemplos de nombres válidos:

```
let userName;
let test123;
```

Cuando el nombre contiene varias palabras, se suele usar el estilo camelCase (capitalización en camello), donde las palabras van pegadas una detrás de otra, con cada inicial en mayúscula: miNombreMuyLargo.

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.

Los siguientes nombres son válidos:

```
let $ = 1; // Declara una variable con el nombre "$"
let _ = 2; // y ahora una variable con el nombre "_"
alert($ + _); // 3
```

5.4.Nombres reservados

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.

5.5. Una asignación sin utilizar use strict

Normalmente, debemos definir una variable antes de utilizarla. Pero, en los viejos tiempos, era técnicamente posible crear una variable simplemente asignando un valor sin utilizar 'let'. Esto aún funciona si no ponemos 'use strict' en nuestros scripts para mantener la compatibilidad con scripts antiguos.

```
// nota: no se utiliza "use strict" en este ejemplo

num = 5; // se crea la variable "num" si no existe antes

alert(num); // 5
```

Esto es una mala práctica que causaría errores en 'strict mode':

```
"use strict";

num = 5; // error: num no está definida
```

5.6. Constantes

Para declarar una variable constante (inmutable) use const en vez de let:

```
const myBirthday = '18.04.1982';
```

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!
```

Cuando un programador está seguro de que una variable nunca cambiará, puede declarar la variable con const para garantizar y comunicar claramente este hecho a todos.

5.7. 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:

- COLOR_ORANGE es mucho más fácil de recordar que "#FF7F00".
- Es mucho más fácil escribir mal "#FF7F00" que COLOR_ORANGE.
- Al leer el código, COLOR_ORANGE tiene mucho más significado que #FF7F00.

5.8. Tipos de datos

Un valor en JavaScript siempre pertenece a un tipo de dato determinado. Por ejemplo, un string o un número.

Hay ocho tipos de datos básicos en JavaScript. Podemos almacenar un valor de cualquier tipo dentro de una variable. Por ejemplo, una variable puede contener en un momento un string y luego almacenar un número:

```
// no hay error
let message = "hola";
message = 123456;
```

Los lenguajes de programación que permiten estas cosas, como JavaScript, se denominan “dinámicamente tipados”, lo que significa que allí hay tipos de datos, pero las variables no están vinculadas rígidamente a ninguno de ellos.

Number

```
let n = 123;
n = 12.345;
```

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.

Infinity representa el Infinito matemático ∞. Es un valor especial que es mayor que cualquier número.

Podemos obtenerlo como resultado de la división por cero:

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

O simplemente hacer referencia a él directamente:

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

NaN representa un error de cálculo. Es el resultado de una operación matemática incorrecta o indefinida, por ejemplo:

```
alert( "no es un número" / 2 ); // NaN, tal división es errónea
```

NaN es “pegajoso”. Cualquier otra operación sobre NaN devuelve NaN:

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

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).

Los valores numéricos especiales pertenecen formalmente al tipo “número”. Por supuesto que no son números en el sentido estricto de la palabra.

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.

BigInt se agregó recientemente al lenguaje para representar enteros de longitud arbitraria.

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

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

String

Un string en JavaScript es una cadena de caracteres y debe colocarse entre comillas.

```
let str = "Hola";
let str2 = 'Las comillas simples también están bien';
let phrase = `se puede incrustar otro ${str}`;
```

En JavaScript, hay 3 tipos de comillas.

- Comillas dobles: "Hola".
- Comillas simples: 'Hola'.
- 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
```

La expresión dentro de ${...} se evalúa y el resultado pasa a formar parte de la cadena. Podemos poner cualquier cosa ahí dentro: una variable como name, una expresión aritmética como 1 + 2, o algo más complejo.

Toma en cuenta que esto sólo se puede hacer con los backticks. ¡Las otras comillas no tienen esta capacidad de incrustación!

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
```

Los valores booleanos también son el resultado de comparaciones:

```
let isGreater = 4 > 1;
alert( isGreater ); // verdadero (el resultado de la comparación es "sí")
```

El valor “null”.

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”.

El código anterior indica que el valor de age es desconocido o está vacío por alguna razón.

El valor “undefined” (indefinido)

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:

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

Técnicamente, es posible asignar undefined a cualquier variable:

```
let age = 100;
// cambiando el valor a undefined
age = undefined;
alert(age); // "undefined"
```

…Pero no recomendamos hacer eso. 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.

Object y Symbol

El tipo object (objeto) es especial.

Todos los demás tipos se llaman “primitivos” porque sus valores pueden contener una sola cosa (ya sea una cadena, un número o lo que sea). Por el contrario, los objetos se utilizan para almacenar colecciones de datos y entidades más complejas.

El tipo symbol (símbolo) se utiliza para crear identificadores únicos para los objetos. Tenemos que mencionarlo aquí para una mayor integridad, pero es mejor estudiar este tipo después de los objetos.

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:

- Math es un objeto incorporado que proporciona operaciones matemáticas.
- 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. Definitivamente null no es un objeto. Es un valor especial con un tipo propio separado.
- El resultado de typeof alert es "function" porque alert es una función. Estudiaremos las funciones en los próximos capítulos donde veremos que no hay ningún tipo especial “function” en JavaScript. Las funciones pertenecen al tipo objeto. Técnicamente dicho comportamiento es incorrecto, pero puede ser conveniente en la práctica.

6. INTERACCIÓN

Cubrimos 3 funciones específicas del navegador para interactuar con los usuarios:

6.1.alert

Muestra un mensaje.

```
alert("Hello");
```

La mini ventana con el mensaje se llama * ventana modal *. La palabra “modal” significa que el visitante no puede interactuar con el resto de la página, presionar otros botones, etc., hasta que se haya ocupado de la ventana. En este caso, hasta que presionen “OK”.

6.2.prompt

Muestra un mensaje pidiendo al usuario que introduzca un texto. Retorna el texto o, si se hace clic en CANCELAR o se presiona Esc, retorna null.

```
result = prompt(title, [default]);
```

6.3.confirm

muestra un mensaje y espera a que el usuario pulse “OK” o “CANCELAR”. Retorna true si se presiona OK y false si se presiona CANCEL/Esc.

Todos estos métodos son modales: detienen la ejecución del script y no permiten que el usuario interactúe con el resto de la página hasta que la ventana se haya cerrado.

Hay dos limitaciones comunes a todos los métodos anteriores:

- La ubicación exacta de la ventana modal está determinada por el navegador. Normalmente, está en el centro.
- El aspecto exacto de la ventana también depende del navegador. No podemos modificarlo.

Ese es el precio de la simplicidad. Existen otras formas de mostrar ventanas más atractivas e interactivas para el usuario, pero si la apariencia no importa mucho, estos métodos funcionan bien.

7. CONVERSIONES DE TIPOS

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.

7.1. 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:

```
let value = true;
alert(typeof value); // boolean
value = String(value); // ahora value es el string "true"
alert(typeof value); // string
```

La conversión a string es bastante obvia. El boolean false se convierte en "false", null en "null", etc.

7.2. 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
```

La conversión explícita es requerida usualmente cuando leemos un valor desde una fuente basada en texto, como lo son los campos de texto en los formularios, pero que esperamos que contengan un valor numérico.

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
```

Adición '+' concatena strings

Casi todas las operaciones matemáticas convierten valores a números. Una excepción notable es la suma +. Si uno de los valores sumados es un string, el otro valor es convertido a string.

Luego, los concatena (une):

```
alert( 1 + '2' ); // '12' (string a la derecha)
alert( '1' + 2 ); // '12' (string a la izquierda)
```

Esto ocurre solo si al menos uno de los argumentos es un string, en caso contrario los valores son convertidos a número.

7.3. ToBoolean

La conversión a boolean es la más simple.

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:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false
alert( Boolean("hola") ); // true
alert( Boolean("") ); // false
Ten en cuenta: el string con un cero "0" es true
Algunos lenguajes (como PHP) tratan "0" como false. Pero en JavaScript, un string no vacío es siempre true.
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // sólo espacios, también true
```

8. OPERADORES BÁSICOS Y MATEMÁTICOS

8.1. Términos: “unario”, “binario”, “operando”

Antes de continuar, comprendamos la terminología común.

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
```

Formalmente, estamos hablando de dos operadores distintos: la negación unaria (un operando: revierte el símbolo) y la resta binaria (dos operandos: resta).

8.2. Matemáticas

Están soportadas las siguientes operaciones:

- Suma +,
- Resta -,
- Multiplicación *,
- División /,
- Resto %,
- Exponenciación **.

Los primeros cuatro son conocidos mientras que % y ** deben ser explicados más ampliamente.

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:

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

Exponenciación **

El operador exponenciación a ** b eleva a a la potencia de b.

En matemáticas de la escuela, lo escribimos como ab.

Por ejemplo:

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

Matemáticamente, la exponenciación está definida para operadores no enteros también.

Por ejemplo, la raíz cuadrada es el exponente ½:

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

Concatenación de cadenas con el binario +

Ahora veamos las características de los operadores de JavaScript que van más allá de la aritmética escolar.

Normalmente el operador + suma números.

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

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

Tenga presente que si uno de los operandos es una cadena, el otro es convertido a una cadena también.

Por ejemplo:

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

Vieron, no importa si el primer operando es una cadena o el segundo.

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.

Desde la escuela, todos sabemos que la multiplicación en la expresión 1 + 2 * 2 debe calcularse antes de la suma. Eso es exactamente la precedencia. Se dice que la multiplicación tiene una mayor precedencia que la suma.

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

Tengamos en cuenta que una asignación = también es un operador. Está listado en la tabla de precedencia con la 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
```

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;
```

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

```
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
```

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:

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

- Decremento -- decrementa una variable por 1:

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

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 (>>>)

Estos operadores se usan muy raramente, cuando necesitamos manejar la representación de números en su más bajo nivel. No tenemos en vista usarlos pronto pues en el desarrollo web tiene poco uso; pero en ciertas áreas especiales, como la criptografía, son útiles. Puedes leer el artículo Operadores a nivel de bit en MDN cuando surja la necesidad.

9. COMPARACIONES

Conocemos muchos operadores de comparación de las matemáticas:

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.

9.1. 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:

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

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

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

9.2. 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:

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

El algoritmo para comparar dos cadenas es simple:

Compare el primer carácter de ambas cadenas.

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.

De lo contrario, si los primeros caracteres de ambas cadenas son los mismos, compare los segundos caracteres de la misma manera.

Repita hasta el final de cada cadena.

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.

9.3.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
```

9.4.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.

9.5.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.

```
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.

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

10. CONDICIONALES

10.1. La 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:

```
let year = prompt('¿En que año fué publicada la especificación ECMAScript?', '');
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:

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

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.

Entonces, el código bajo esta condición nunca se ejecutaría:

```
if (0) { // 0 es falso
  ...
}
```

…y dentro de esta condición siempre se ejecutará:

```
if (1) { // 1 es verdadero
  ...
}
```

10.2. 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?', '');
if (year == 2015) {
  alert( '¡Lo adivinaste, correcto!' );
} else {
  alert( '¿Cómo puedes estar tan equivocado?' ); // cualquier valor excepto 2015
}
```

10.3. Muchas condiciones: “else if”

A veces queremos probar más de una condición. La clausula else if nos permite hacer esto.

Por ejemplo:

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


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.

10.4. 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;
```

11. OPERADORES LÓGICOS

Hay cuatro operadores lógicos en JavaScript: || (O), && (Y), ! (NO).

Aunque sean llamados lógicos, pueden ser aplicados a valores de cualquier tipo, no solo booleanos. El resultado también puede ser de cualquier tipo.

11.1. || (OR)

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

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

En la programación clásica, el OR lógico esta 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.' );
}
```

11.2.&& (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" );
}
```

11.3. ! (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 mas prolija de realizar lo mismo – una función integrada Boolean:

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

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

12. BUCLES: WHILE Y FOR

Usualmente necesitamos repetir acciones.

Por ejemplo, mostrar los elementos de una lista uno tras otro o simplemente ejecutar el mismo código para cada número del 1 al 10.

Los Bucles son una forma de repetir el mismo código varias veces.

Los bucles for…of y for…in

12.1. El bucle “while”

El bucle while (mientras) tiene la siguiente sintaxis:

```
while (condition) {
  // código
  // llamado "cuerpo del bucle"
}
```

Mientras la condición condition sea verdadera, el código del cuerpo del bucle será ejecutado.

Por ejemplo, el bucle debajo imprime i mientras se cumpla i < 3:

```
let i = 0;
while (i < 3) { // muestra 0, luego 1, luego 2
  alert( i );
  i++;
}
```

Cada ejecución del cuerpo del bucle se llama iteración. El bucle en el ejemplo de arriba realiza 3 iteraciones.

Si faltara i++ en el ejemplo de arriba, el bucle sería repetido (en teoría) eternamente. En la práctica, el navegador tiene maneras de detener tales bucles desmedidos; y en el JavaScript del lado del servidor, podemos eliminar el proceso.

Cualquier expresión o variable puede usarse como condición del bucle, no solo las comparaciones: El while evaluará y transformará la condición a un booleano.

12.2. El bucle "do...while"

La comprobación de la condición puede ser movida debajo del cuerpo del bucle usando la sintaxis do..while:

```
do {
  // cuerpo del bucle
} while (condition);
```

El bucle primero ejecuta el cuerpo, luego comprueba la condición, y, mientras sea un valor verdadero, la ejecuta una y otra vez.

Por ejemplo:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Esta sintaxis solo debe ser usada cuando quieres que el cuerpo del bucle sea ejecutado al menos una vez sin importar que la condición sea verdadera. Usualmente, se prefiere la otra forma: while(...) {...}.

12.3. El bucle "for"

El bucle for es más complejo, pero también el más usado.

Se ve así:

```
for (begin; condition; step) { // (comienzo, condición, paso)
  // ... cuerpo del bucle ...
}
```

Aprendamos el significado de cada parte con un ejemplo. El bucle debajo corre alert(i) para i desde 0 hasta (pero no incluyendo) 3:

```
for (let i = 0; i < 3; i++) { // muestra 0, luego 1, luego 2
  alert(i);
}
```

Declaración de variable en línea

Aquí, la variable “counter” i es declarada en el bucle. Esto es llamado una declaración de variable “en línea”. Dichas variables son visibles solo dentro del bucle.

```
for (let i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // error, no existe dicha variable
```

En vez de definir una variable, podemos usar una que ya exista:

```
let i = 0;
for (i = 0; i < 3; i++) { // usa una variable existente
  alert(i); // 0, 1, 2
}
alert(i); // 3, visible, porque fue declarada fuera del bucle
```

12.4. Rompiendo el bucle

Normalmente, se sale de un bucle cuando la condición se vuelve falsa.

Pero podemos forzar una salida en cualquier momento usando la directiva especial break.

Por ejemplo, el bucle debajo le pide al usuario por una serie de números, “rompiéndolo” cuando un número no es ingresado:

```
let sum = 0;
while (true) {
  let value = +prompt("Ingresa un número", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Suma: ' + sum );
```

La directiva break es activada en la línea (*) si el usuario ingresa una línea vacía o cancela la entrada. Detiene inmediatamente el bucle, pasando el control a la primera línea después del bucle. En este caso, alert.

La combinación “bucle infinito + break según sea necesario” es ideal en situaciones donde la condición del bucle debe ser comprobada no al inicio o al final de el bucle, sino a la mitad o incluso en varias partes del cuerpo.

12.5. Continuar a la siguiente iteración

La directiva continue es una "versión más ligera" de break. No detiene el bucle completo. En su lugar, detiene la iteración actual y fuerza al bucle a comenzar una nueva (si la condición lo permite).

Podemos usarlo si hemos terminado con la iteración actual y nos gustaría movernos a la siguiente.

El bucle debajo usa continue para mostrar solo valores impares:

```
for (let i = 0; i < 10; i++) {
 // si es verdadero, saltar el resto del cuerpo
 if (i % 2 == 0) continue;
 alert(i); // 1, luego 3, 5, 7, 9
}
```

Para los valores pares de i, la directiva continue deja de ejecutar el cuerpo y pasa el control a la siguiente iteración de for (con el siguiente número). Así que el alert solo es llamado para valores impares.

13. LA SENTENCIA "SWITCH"

Una sentencia switch puede reemplazar múltiples condiciones if.

Provee una mejor manera de comparar un valor con múltiples variantes.

13.1. La sintaxis

switch tiene uno o mas bloques casey un opcional default.

Se ve de esta forma:

```
switch(x) {
  case 'valor1':  // if (x === 'valor1')
    ...
    [break]
  case 'valor2':  // if (x === 'valor2')
    ...
    [break]
  default:
    ...
    [break]
}
```

El valor de x es comparado contra el valor del primer case (en este caso, valor1), luego contra el segundo (valor2) y así sucesivamente, todo esto bajo una igualdad estricta.

Si la igualdad es encontrada, switch empieza a ejecutar el código iniciando por el primer case correspondiente, hasta el break más cercano (o hasta el final del switch).

Si no se cumple ningún caso entonces el código default es ejecutado (si existe).

Ejemplo

```
let a = 2 + 2;
switch (a) {
 case 3:
  alert( 'Muy pequeño' );
  break;
 case 4:
  alert( '¡Exacto!' );
  break;
 case 5:
  alert( 'Muy grande' );
  break;
 default:
  alert( "Desconozco estos valores" );
}
```

Aquí el switch inicia comparando a con la primera variante case que es 3. La comparación falla.

Luego 4. La comparación es exitosa, por tanto la ejecución empieza desde case 4 hasta el break más cercano.

Si no existe break entonces la ejecución continúa con el próximo case sin ninguna revisión.

Un ejemplo sin break:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Muy pequeño' );
  case 4:
    alert( '¡Exacto!' );
  case 5:
    alert( 'Muy grande' );
  default:
    alert( "Desconozco estos valores" );
}
```

En el ejemplo anterior veremos ejecuciones de tres alert secuenciales:

```
alert( '¡Exacto!' );
alert( 'Muy grande' );
alert( "Desconozco estos valores" );
```

13.2. Agrupamiento de “case”

Varias variantes de case los cuales comparten el mismo código pueden ser agrupadas.

Por ejemplo, si queremos que se ejecute el mismo código para case 3 y case 5:

```
let a = 2 + 2;
switch (a) {
 case 4:
  alert('¡Correcto!');
  break;
 case 3:                  // (*) agrupando dos cases
 case 5:
  alert('¡Incorrecto!');
  alert("¿Por qué no tomas una clase de matemáticas?");
  break;
 default:
  alert('El resultado es extraño. Realmente.');
}
```

Ahora ambos, 3 y 5, muestran el mismo mensaje.

La capacidad de “agrupar” los case es un efecto secundario de cómo trabaja switch/case sin break. Aquí la ejecución de case 3 inicia desde la línea (*) y continúa a través de case 5, porque no existe break.

14. FUNCIONES

Muy a menudo necesitamos realizar acciones similares en muchos lugares del script.

Por ejemplo, debemos mostrar un mensaje atractivo cuando un visitante inicia sesión, cierra sesión y tal vez en otros momentos.

Las funciones son los principales "bloques de construcción" del programa. Permiten que el código se llame muchas veces sin repetición.

Ya hemos visto ejemplos de funciones integradas, como alert(message), prompt(message, default) y confirm(question). Pero también podemos crear funciones propias.

14.1. Declaración de funciones

Para crear una función podemos usar una declaración de función.

Se ve como aquí:

```
function showMessage() {
  alert( '¡Hola a todos!' );
}
```

La palabra clave function va primero, luego va el nombre de función, luego una lista de parámetros entre paréntesis (separados por comas, vacía en el ejemplo anterior) y finalmente el código de la función entre llaves, también llamado "el cuerpo de la función".

```
function name(parameter1, parameter2, ... parameterN) {
 // body
}
```

Nuestra nueva función puede ser llamada por su nombre: showMessage().

Por ejemplo:

```
function showMessage() {
  alert( '¡Hola a todos!' );
}
showMessage();
showMessage();
```

La llamada showMessage() ejecuta el código de la función. Aquí veremos el mensaje dos veces.

Este ejemplo demuestra claramente uno de los propósitos principales de las funciones: evitar la duplicación de código…

14.2. Variables Locales

Una variable declarada dentro de una función solo es visible dentro de esa función.

Por ejemplo:

```
function showMessage() {
  let message = "Hola, ¡Soy JavaScript!"; // variable local
  alert( message );
}
showMessage(); // Hola, ¡Soy JavaScript!
alert( message ); // <-- ¡Error! La variable es local para esta función
```

14.3. Variables Externas

Una función también puede acceder a una variable externa, por ejemplo:

```
let userName = 'Juan';
function showMessage() {
  let message = 'Hola, ' + userName;
  alert(message);
}
showMessage(); // Hola, Juan
```

La función tiene acceso completo a la variable externa. Puede modificarlo también.

Por ejemplo:

```
let userName = 'Juan';
function showMessage() {
  userName = "Bob"; // (1) Cambió la variable externa
  let message = 'Hola, ' + userName;
  alert(message);
}
alert( userName ); // Juan antes de llamar la función
showMessage();
alert( userName ); // Bob, el valor fué modificado por la función
```

14.4. Parámetros

Podemos pasar datos arbitrarios a funciones usando parámetros.

En el siguiente ejemplo, la función tiene dos parámetros: from y text.

```
function showMessage(from, text) { // parámetros: from, text
  alert(from + ': ' + text);
}
showMessage('Ann', '¡Hola!'); // Ann: ¡Hola! (*)
showMessage('Ann', "¿Cómo estás?"); // Ann: ¿Cómo estás? (**)
```

Cuando la función se llama (*) y (**), los valores dados se copian en variables locales from y text. Y la función las utiliza.

Aquí hay un ejemplo más: tenemos una variable from y la pasamos a la función. Tenga en cuenta: la función cambia from, pero el cambio no se ve afuera, porque una función siempre obtiene una copia del valor:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // hace que "from" se vea mejor
  alert( from + ': ' + text );
}
let from = "Ann";
showMessage(from, "Hola"); // *Ann*: Hola


// el valor de "from" es el mismo, la función modificó una copia local
alert( from ); // Ann
```

Cuando un valor es pasado como un parámetro de función, también se denomina argumento.

14.5. Valores predeterminados

Si una función es llamada, pero no se le proporciona un argumento, su valor correspondiente se convierte en undefined.

Por ejemplo, la función mencionada anteriormente showMessage(from, text) se puede llamar con un solo argumento:

```
showMessage("Ann");
```

Eso no es un error. La llamada mostraría "Ann: undefined". Como no se pasa un valor de text, este se vuelve undefined.

Podemos especificar un valor llamado “predeterminado” o “por defecto” (es el valor que se usa si el argumento fue omitido) en la declaración de función usando =:

```
function showMessage(from, text = "sin texto") {
  alert( from + ": " + text );
}
showMessage("Ann"); // Ann: sin texto
```

Ahora, si no se pasa el parámetro text, obtendrá el valor "sin texto"

El valor predeterminado también se asigna si el parámetro existe pero es estrictamente igual a undefined:

```
showMessage("Ann", undefined); // Ann: sin texto
```

Aquí "sin texto" es un string, pero puede ser una expresión más compleja, la cual solo es evaluada y asignada si el parámetro falta. Entonces, esto también es posible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() solo se ejecuta si text no fue asignado
  // su resultado se convierte en el valor de texto
}
```

14.6. Parámetros predeterminados alternativos

A veces tiene sentido asignar valores predeterminados a los parámetros más tarde, después de la declaración de función.

Podemos verificar si un parámetro es pasado durante la ejecución de la función comparándolo con undefined:

```
function showMessage(text) {
 // ...
 if (text === undefined) { // si falta el parámetro
   text = 'mensaje vacío';
 }
 alert(text);
}
showMessage(); // mensaje vacío
```

…O podemos usar el operador ||:

```
function showMessage(text) {
 // si text es indefinida o falsa, la establece a 'vacío'
 text = text || 'vacío';
 ...
}
```

14.7. Devolviendo un valor

Una función puede devolver un valor al código de llamada como resultado.

El ejemplo más simple sería una función que suma dos valores:

```
function sum(a, b) {
  return a + b;
}
let result = sum(1, 2);
alert( result ); // 3
```

La directiva return puede estar en cualquier lugar de la función. Cuando la ejecución lo alcanza, la función se detiene y el valor se devuelve al código de llamada (asignado al result anterior).

Puede haber muchos return en una sola función. Por ejemplo:

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('¿Tienes permiso de tus padres?');
  }
}
let age = prompt('¿Qué edad tienes?', 18);
if ( checkAge(age) ) {
  alert( 'Acceso otorgado' );
} else {
  alert( 'Acceso denegado' );
}
```

Es posible utilizar return sin ningún valor. Eso hace que la función salga o termine inmediatamente.

Por ejemplo:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }
  alert( "Mostrándote la película" ); // (*)
  // ...
}
```

En el código de arriba, si checkAge(age) devuelve false, entonces showMovie no mostrará la alert.

Una función con un return vacío, o sin return, devuelve undefined

Si una función no devuelve un valor, es lo mismo que si devolviera undefined:

```
function doNothing() { /* empty */ }
alert( doNothing() === undefined ); // true
```

Un return vacío también es lo mismo que return undefined:

```
function doNothing() {
  return;
}
alert( doNothing() === undefined ); // true
```