UT2 Desarrollo web

API: **interfaces de programación de aplicaciones** (**API**) le brindan superpoderes adicionales para usar en su código JavaScript.

Las API del navegador están integradas en su navegador web y pueden exponer datos del entorno informático circundante o hacer cosas útiles y complejas**. Por ejemplo:**

Le **DOM (Document Object Model) API** permite manipular HTML y CSS, crear, eliminar y cambiar HTML, aplicar dinámicamente nuevos estilos a su página, etc. Cada vez que vea aparecer una ventana emergente en una página, o que se muestre algún contenido nuevo (como vimos arriba en nuestra demostración simple) por ejemplo, ese es el DOM en acción.

El Geolocation API recupera información geográfica. Así es como Google Maps puede encontrar su ubicación y trazarla en un mapa.

Las **API Canvas y WebGLle** permiten crear gráficos animados en 2D y 3D.

A las API de audio y video les gusta **HTMLMediaElementy WebRTCle** permiten hacer cosas realmente interesantes con multimedia, como reproducir audio y video directamente en una página web, o tomar un video de su cámara web y mostrarlo en la computadora de otra persona

Las API de terceros no están integradas en el navegador de forma predeterminada y, por lo general, debe obtener su código e información de algún lugar de la Web.

Cada pestaña del navegador tiene su propio cubo separado para ejecutar código (estos cubos se denominan "entornos de ejecución" en términos técnicos); esto significa que, en la mayoría de los casos, el código de cada pestaña se ejecuta completamente por separado y el código de una pestaña no puede ejecutarse directamente. afectar el código en otra pestaña, o en otro sitio web. Esta es una buena medida de seguridad

Cuando el navegador encuentra un bloque de JavaScript, generalmente lo ejecuta en orden, de arriba a abajo. Esto significa que debe tener cuidado con el orden en que coloca las cosas

Es posible que escuche los términos **interpretados** y **compilados** en el contexto de la programación. En los lenguajes interpretados, el código se ejecuta de arriba a abajo y el resultado de la ejecución del código se devuelve inmediatamente. No tiene que transformar el código en una forma diferente antes de que el navegador lo ejecute. El código se recibe en su forma de texto amigable para el programador y se procesa directamente a partir de ahí.

Los lenguajes compilados, por otro lado, se transforman (compilan) en otra forma antes de que la computadora los ejecute. Por ejemplo, C/C++ se compilan en un código de máquina que luego ejecuta la computadora. El programa se ejecuta desde un formato binario, que se generó a partir del código fuente del programa original.

JavaScript es un lenguaje de programación ligero e interpretado. El navegador web recibe el código JavaScript en su forma de texto original y ejecuta el script a partir de ahí.

Código dinámico versus estático

La palabra **dinámica** se usa para describir tanto el JavaScript del lado del cliente como los lenguajes del lado del servidor; se refiere a la capacidad de actualizar la visualización de una página web/aplicación para mostrar diferentes cosas en diferentes circunstancias, generando nuevo contenido según sea necesario.

Una página web sin contenido que se actualiza dinámicamente se denomina **estática** ; simplemente muestra el mismo contenido todo el tiempo.

Para agregar JavaScript a la pagina se hace en el documento html de esta forma (JavaScript **interno**):

```
<script>
  // JavaScript goes here
</script>
```

Para agregar JavaScript a la pagina de **forma externa** (Desde un archivo externo y no dentro del html) se hace así:

1. Primero, cree un nuevo archivo en el mismo directorio que su archivo HTML de muestra. Llámelo `script.js`: asegúrese de que tenga esa extensión de nombre de archivo .js, ya que así es como se reconoce como JavaScript.
2. Reemplace su `<script>` elemento actual con lo siguiente:

```
<script src="script.js" defer></script>
```

Es una mala práctica contaminar su HTML con JavaScript

En los ejemplos de código anteriores, en los ejemplos internos y externos, JavaScript se carga y ejecuta en el encabezado del documento, antes de que se analice el cuerpo HTML. Esto podría causar un error, por lo que hemos usado algunas construcciones para evitarlo.

En el ejemplo interno, puedes ver esta estructura alrededor del código:

```
document.addEventListener("DOMContentLoaded", () => {
  // …
});
```

Este es un detector de eventos, que escucha el `DOMContentLoaded` evento del navegador, lo que significa que el cuerpo HTML está completamente cargado y analizado. El JavaScript dentro de este bloque no se ejecutará hasta que se active ese evento, por lo tanto, se evita el error (aprenderá sobre los eventos más adelante en el curso).

En el ejemplo externo, usamos una característica de JavaScript más moderna para resolver el problema, el `defer` atributo, que le dice al navegador que continúe descargando el contenido HTML una vez que `<script>` se haya alcanzado el elemento de la etiqueta.

```
<script src="script.js" defer></script>
```

En este caso, tanto el script como el HTML se cargarán simultáneamente y el código funcionará.

Usando addEventListener en su lugar

En lugar de incluir JavaScript en su HTML, use una construcción de JavaScript puro. La `querySelectorAll()` función le permite seleccionar todos los botones de una página. Luego puede recorrer los botones, asignando un controlador para cada uno usando `addEventListener()`. El código para esto se muestra a continuación:

```
const buttons = document.querySelectorAll("button");

for (const button of buttons) {
  button.addEventListener("click", createParagraph);
}
```

Esto puede ser un poco más largo que el `onclick` atributo, pero funcionará para todos los botones, sin importar cuántos haya en la página, ni cuántos se agreguen o eliminen. No es necesario cambiar el JavaScript.

Características modernas para evitar problema de script se cargue antes que html Async y defer

asíncrono y diferido

En realidad, hay dos características modernas que podemos usar para evitar el problema del script de bloqueo: `async` y `defer` (que vimos anteriormente). Veamos la diferencia entre estos dos.

Los scripts cargados con el `async` atributo descargarán el script sin bloquear la página mientras se recupera el script. Sin embargo, una vez que se complete la descarga, se ejecutará el script, lo que bloquea la representación de la página. No obtiene ninguna garantía de que los scripts se ejecutarán en un orden específico. Es mejor utilizarlo `async` cuando los scripts de la página se ejecutan de forma independiente y no dependen de ningún otro script de la página.

Los scripts cargados con el `defer` atributo (ver más abajo) se ejecutarán en el orden en que aparecen en la página y se ejecutarán tan pronto como se descarguen el script y el contenido:

```
<script defer src="js/vendor/jquery.js"></script>

<script defer src="js/script2.js"></script>

<script defer src="js/script3.js"></script>
```

En el segundo ejemplo, podemos estar seguros de que `jquery.js` cargará antes `script2.js` y `script3.js` que `script2.js` cargará antes `script3.js`. No se ejecutarán hasta que el contenido de la página se haya cargado por completo, lo cual es útil si sus scripts dependen de que el DOM esté en su lugar (por ejemplo, modifican uno o más elementos en la página).

`async` debe usarse cuando tiene un montón de scripts de fondo para cargar, y solo quiere tenerlos en su lugar lo antes posible. Por ejemplo, tal vez tenga algunos archivos de datos del juego para cargar, que serán necesarios cuando el juego realmente comience, pero por ahora solo quiere seguir mostrando la introducción del juego, los títulos y el lobby, sin que se bloqueen con la carga del script.

Para resumir:

- `async` y `defer` ambos le indican al navegador que descargue los scripts en un hilo separado, mientras que el resto de la página (el DOM, etc.) se descarga, por lo que la carga de la página no se bloquea durante el proceso de obtención.
- los scripts con un `async` atributo se ejecutarán tan pronto como se complete la descarga. Esto bloquea la página y no garantiza ningún orden de ejecución específico.
- los scripts con un `defer` atributo se cargarán en el orden en que están y solo se ejecutarán una vez que todo haya terminado de cargarse.
- Si sus scripts deben ejecutarse inmediatamente y no tienen ninguna dependencia, entonces use `async`.
- Si sus secuencias de comandos deben esperar a que se analicen y dependen de otras secuencias de comandos y/o del DOM, cárguelas usando `defer` y coloque sus elementos correspondientes `<script>` en el orden en que desea que el navegador los ejecute.

Comentarios:

- Un comentario de una sola línea se escribe después de una barra inclinada doble (//), por ejemplo

```
// I am a comment
```

- Un comentario de varias líneas se escribe entre las cadenas /* y */, por ejemplo

```
/*
  I am also
  a comment
*/
```

Tipos de datos:

Escritura dinámica

JavaScript es un lenguaje dinámico con tipos dinámicos. Las variables en JavaScript no están asociadas directamente con ningún tipo de valor en particular, y a cualquier variable se le pueden asignar (y reasignar) valores de todos los tipos:

```
let foo = 42; // foo is now a number
foo = "bar"; // foo is now a string
foo = true; // foo is now a boolean
```

Todos los tipos primitivos, excepto `null`, pueden ser probados por el `typeof` operador. `typeof null` devuelve `"object"`, por lo que uno tiene que usar `=== null` para probar `null`.

tipo BigInt

El `BigInt` tipo es una primitiva numérica en JavaScript que puede representar números enteros con una magnitud arbitraria. Con BigInts, puede almacenar y operar de manera segura con números enteros grandes incluso más allá del límite de número entero seguro (`Number.MAX_SAFE_INTEGER`) para Números.

Un BigInt se crea agregando `n` al final de un número entero o llamando a la `BigInt()` función.

Este ejemplo demuestra dónde incrementar el `Number.MAX_SAFE_INTEGER` devuelve el resultado esperado:

```
// BigInt
const x = BigInt(Number.MAX_SAFE_INTEGER); // 9007199254740991n
x + 1n === x + 2n; // false because 9007199254740992n and 9007199254740993n are unequal

// Number
Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2; // true because both are
9007199254740992
```

Las Cadenas en JavaScript son **inmutables**

Las cadenas de JavaScript son inmutables. Esto significa que una vez que se crea una cadena, no es posible modificarla. Los métodos de cadena crean nuevas cadenas basadas en el contenido de la cadena actual, por ejemplo:

- Una subcadena del original usando `substring()`.
- Una concatenación de dos cadenas mediante el operador de concatenación (`+`) o `concat()`.

Puede ser tentador usar cadenas para representar datos complejos. Hacer esto viene con beneficios a corto plazo:

- Es fácil construir cadenas complejas con concatenación.
- Las cadenas son fáciles de depurar (lo que ve impreso es siempre lo que está en la cadena).
- Las cadenas son el denominador común de muchas API (campos de entrada , valores de almacenamiento local , `XMLHttpRequest` respuestas al usar `responseText` , etc.) y puede ser tentador trabajar solo con cadenas.

Con convenciones, es posible representar cualquier estructura de datos en una cadena. Esto no lo

Hay dos tipos de propiedades de objeto: la propiedad ***de datos*** y la propiedad ***de acceso*** . Cada propiedad tiene *atributos* correspondientes . El motor de JavaScript accede internamente a cada atributo, pero puede configurarlos Object.defineProperty()o leerlos Object.getOwnPropertyDescriptor().

propiedad de datos

Las propiedades de datos asocian una clave con un valor. Puede ser descrito por los siguientes atributos:

`value`

El valor recuperado por un acceso de obtención de la propiedad. Puede ser cualquier valor de JavaScript.

`writable`

Un valor booleano que indica si la propiedad se puede cambiar con una asignación.

`enumerable`

Un valor booleano que indica si la propiedad se puede enumerar mediante un `for...in` bucle. Consulte también Enumerabilidad y propiedad de las propiedades para conocer cómo la enumerabilidad interactúa con otras funciones y sintaxis.

`configurable`

Un valor booleano que indica si la propiedad se puede eliminar, se puede cambiar a una propiedad de acceso y se pueden cambiar sus atributos.

Propiedad de acceso

Asocia una tecla con una de las dos funciones de acceso (`get` y `set`) para recuperar o almacenar un valor.

> **Nota:** es importante reconocer su *propiedad* de acceso , no *el método* de acceso . Podemos dar accesos similares a una clase a un objeto de JavaScript usando una función como un valor, pero eso no convierte al objeto en una clase.

Una propiedad de acceso tiene los siguientes atributos:

`get`

Una función llamada con una lista de argumentos vacía para recuperar el valor de la propiedad cada vez que se realiza un acceso al valor. Véase también captadores . Puede `undefined` ser

`set`

Una función llamada con un argumento que contiene el valor asignado. Se ejecuta cada vez que se intenta cambiar una propiedad especificada. Véase también setters . Puede `undefined` ser

`enumerable`

Un valor booleano que indica si la propiedad se puede enumerar mediante un `for...in` bucle. Consulte también Enumerabilidad y propiedad de las propiedades para conocer cómo la enumerabilidad interactúa con otras funciones y sintaxis.

`configurable`

Un valor booleano que indica si la propiedad se puede eliminar, se puede cambiar a una propiedad de datos y se pueden cambiar sus atributos.

Variables

Una variable es un contenedor para un valor, como un número que podríamos usar en una suma, o una cadena que podríamos usar como parte de una oración. Pero una cosa especial acerca de las variables es que los valores que contienen pueden cambiar.

Declarar una variable

Para usar una variable, primero debes crearla — precisamente, a esto lo llamamos declarar la variable. Para hacerlo, escribimos la palabra clave `var` o `let` seguida del nombre con el que deseas llamar a tu variable:

```
let myName;
let myAge;
```

Iniciar una variable

Una vez que hayas declarado una variable, la puedes iniciar con un valor. Para ello, escribe el nombre de la variable, seguido de un signo igual (`=`), seguido del valor que deseas darle. Por ejemplo:

```
myName = 'Chris';
myAge = 37;
```

En segundo lugar, cuando usas `var`, puedes declarar la misma variable tantas veces como desees, pero con `let` no puedes. Lo siguiente funcionaría:

```
var myName = 'Chris';
var myName = 'Bob';
```

Pero lo siguiente arrojaría un error en la segunda línea:

```
let myName = 'Chris';
let myName = 'Bob';
```

Tendrías que hacer esto en su lugar:

```
let myName = 'Chris';
myName = 'Bob';
```

Por estas y otras razones, se recomienda utilizar `let` tanto como sea posible en tu código, en lugar de `var`. No hay ninguna razón para usar `var`, a menos que necesites admitir versiones antiguas de Internet Explorer con tu código (no es compatible con `let` hasta la versión 11; Edge el moderno navegador de Windows admite `let` perfectamente).

Constantes en JavaScript

Muchos lenguajes de programación tienen el concepto de una *constante* — un valor que, una vez declarado no se puede cambiar. Hay muchas razones por las que querrías hacer esto, desde la seguridad (si un script de un tercero cambia dichos valores, podría causar problemas) hasta la depuración y la comprensión del código (es más difícil cambiar accidentalmente valores que no se deben cambiar y estropear cosas claras).

En los primeros días de JavaScript, las constantes no existían. En JavaScript moderno, tenemos la palabra clave `const`, que nos permite almacenar valores que nunca se pueden cambiar:

```
const daysInWeek = 7;
const hoursInDay = 24;
```

`const` funciona exactamente de la misma manera que `let`, excepto que a `const` no le puedes dar un nuevo valor. En el siguiente ejemplo, la segunda línea arrojará un error:

```
const daysInWeek = 7;
daysInWeek = 8;
```

Strings

Comillas simples vs. comillas dobles

1. En JavaScript, puedes escoger entre comillas simple y dobles para envolver tus cadenas. Ambas funcionarán correctamente:

```
var simp = 'Comillas simples.';
var dobl = "Comillas dobles.";
simp;
dobl;
```

4. Sin embargo, no puedes usar el mismo tipo de comillas en el interior de una cadena que ya las tiene en los extremos. Lo siguiente devuelve error, porque confunde al navegador respecto de dónde termina la cadena:

```
var bigmouth = 'I've got no right to take my place...';
```

Escapando caracteres en una cadena

Para solucionar nuestro problema anterior, necesitamos "escapar" el asunto de las comillas. Escapar caracteres significa que les hacemos algo para asegurarnos que sean reconocidos como texto, y no parte del código. En JavaScript, colocamos una barra invertida justo antes del caracter. Intenta ésto:

```
var bigmouth = 'I\'ve got no right to take my place...';
bigmouth;
```

Ahora funciona correctamente. Puedes escapar otros caracteres de la misma forma, ej. `\"`, y hay varios códigos más. Ve a Notación de Escape para más detalles.

Arreglos

Ten en cuenta que `array.length` no necesariamente es el número de elementos del arreglo. Considera lo siguiente:

```
var a = ['dog', 'cat', 'hen'];
a[100] = 'fox';
a.length; // 101
```

Recuerda — la longitud de el arreglo es uno más que el índice más alto.

ES2015 introdujo el bucle más conciso `for...of` para objetos iterables como arreglos:

```
for (const currentValue of a) {
  // Haz algo con currentValue
}
```

También puedes iterar sobre un arreglo utilizando el bucle `for...in`, sin embargo, este no itera sobre los elementos del arreglo, sino los índices del arreglo. Además, si alguien agrega nuevas propiedades a `Array.prototype`, también serán iteradas por dicho bucle. Por lo tanto, este tipo de bucle no se recomienda para arreglos.

Otra forma de iterar sobre un arreglo que se agregó con ECMAScript 5 es `arr.forEach()`:

```
['dog', 'cat', 'hen'].forEach(function(currentValue, index, array) {
  // Hacer algo con currentValue o array[index]
});
```

Estudiar funciones link 5 o ver video

Objetos

Objetos

En programación, un objeto es una estructura de código que modela un objeto de la vida real. Puedes tener un objeto simple que represente una caja y contenga información sobre su ancho, largo y alto, o podrías tener un objeto que represente a una persona y contenga datos sobre su nombre, estatura, peso, qué idioma habla, cómo saludarlo, y más.

Intenta ingresar la siguiente línea en tu consola:

```
let dog = { name : 'Spot', breed : 'Dalmatian' };
```

Para recuperar la información almacenada en el objeto, puedes utilizar la siguiente sintaxis:

```
dog.name
```

```
var persona = {
  nombre: ['Bob', 'Smith'],
  edad: 32,
  genero: 'masculino',
  intereses: ['música', 'esquí'],
  bio: function () {
    alert(this.nombre[0] + '' + this.nombre[1] + ' tiene ' + this.edad + ' años. Le gusta ' +
this.intereses[0] + ' y ' + this.intereses[1] + '.');
  },
  saludo: function() {
    alert('Hola, Soy '+ this.nombre[0] + '. ');
  }
};
```

Notación de punto

Arriba, accediste a las propiedades y métodos del objeto usando **notación de punto (dot notation)**. El nombre del objeto (`persona`) actúa como el **espacio de nombre (namespace)**; al cual se debe ingresar primero para acceder a cualquier elemento **encapsulado** dentro del objeto. A continuación, escribe un punto y luego el elemento al que deseas acceder: puede ser el nombre de una simple propiedad, un elemento de una propiedad de arreglo o una llamada a uno de los métodos del objeto, por ejemplo:

```
persona.edad
persona.intereses[1]
persona.bio()
```

Espacios de nombres secundarios

Incluso es posible hacer que el valor de un miembro del objeto sea otro objeto. Por ejemplo, intenta cambiar el miembro nombre de

```
nombre: ['Bob', 'Smith'],
```

a

```
nombre : {
  pila: 'Bob',
  apellido: 'Smith'
},
```

Aquí estamos creando efectivamente un **espacio de nombre secundario (sub-namespace)**. Esto suena complejo, pero en realidad no es así: para acceder a estos elementos solo necesitas un paso adicional que es encadenar con otro punto al final. Prueba estos:

```
persona.nombre.pila
persona.nombre.apellido
```

Notación de corchetes

Hay otra manera de acceder a las propiedades del objeto, usando la notación de corchetes. En lugar de usar estos:

```
persona.edad
persona.nombre.pila
```

Puedes usar

```
persona['edad']
persona['nombre']['pila']
```

Esto se ve muy similar a cómo se accede a los elementos en un arreglo, y básicamente es lo mismo: en lugar de usar un número de índice para seleccionar un elemento, se esta utilizando el nombre asociado con el valor de cada miembro. No es de extrañar que los objetos a veces se denominen **arreglos asociativos**: asocian cadenas de texto a valores de la misma manera que las arreglos asocian números a valores.

Comparadores de igualdad

Comparadores de igualdad e identidad

Existen cuatro algoritmos de igualdad en ES2015:

- Comparación de Igualdad Abstracta (`==`)
- Comparación de Igualdad Estricta (`===`): utilizada por `Array.prototype.indexOf` , `Array.prototype.lastIndexOf` , y `case` -matching
- SameValueZero: utilizado por los constructores de `%TypedArray%` y `ArrayBuffer` , así como por las operaciones `Map` y `Set` , y también por `String.prototype.includes` y `Array.prototype.includes` desde ES2016
- SameValue: utilizado en el resto de los casos

JavaScript proporciona tres operaciones distintas para comparar la igualdad de dos elementos:

- `===` (en-US) - Igualdad estricta (o "triple igual" o "identidad")
- `==` (en-US) - igualdad débil o relajada ("doble igual")
- `Object.is` proporciona SameValue (nuevo en ES2015).

Igualdad Estricta usando `===`

El operador igualdad estricta compara la igualdad de dos valores. Ninguno de estos valores se convierte de manera implícita antes de ser comparado. Si los valores tienen tipos diferentes son considerados diferentes. Por el contrario, si los valores tienen el mismo tipo y no son números, son considerados iguales si tienen el mismo valor. Finalmente, si ambos valores son números, son considerados iguales si ambos no son NaN y tienen el mismo valor, o si uno es +0 y otro -0.

```
var num = 0;
var obj = new String("0");
var str = "0";
var b = false;

console.log(num === num); // true
console.log(obj === obj); // true
console.log(str === str); // true

console.log(num === obj); // false
console.log(num === str); // false
console.log(obj === str); // false
console.log(null === undefined); // false
console.log(obj === null); // false
console.log(obj === undefined); // false
```

Igualdad débil usando ==

El operador igualdad débil compara la igualdad de dos valores después de convertir ambos valores a un tipo de datos común. Tras la conversión , la comparación que se lleva a cabo funciona exactamente como ===. La igual débil es una igualdad simétrica: A == B tiene una semática idéntica a B == A para cualquier valor que tengan A y B (excepto por el orden de las .conversiones de tipo aplicadas)

```
var num = 0;
var obj = new String("0");
var str = "0";
var b = false;

console.log(num == num); // true
console.log(obj == obj); // true
console.log(str == str); // true

console.log(num == obj); // true
console.log(num == str); // true
console.log(obj == str); // true
console.log(null == undefined); // true

// both false, except in rare cases
console.log(obj == null);
console.log(obj == undefined);
```

Igualdad Same-value

La igualdad Same-value se encarga de un último caso de uso: determinar si dos valores son *funcionalmente idénticos* en todos los contextos. (Este caso de uso es un caso de ejemplo del Liskov substitution principle.) Un ejemplo de esto ocurre cuando se intenta hacer mutable una propiedad inmutable.

```
// Add an immutable NEGATIVE_ZERO property to the Number constructor.
Object.defineProperty(Number, "NEGATIVE_ZERO",
                      { value: -0, writable: false, configurable: false, enumerable: false });

function attemptMutation(v)
{
  Object.defineProperty(Number, "NEGATIVE_ZERO", { value: v });
}
```

`Object.defineProperty` que lanzará una excepción cuando se intente cambiar una propiedad inmutable la cambiará, pero no hará nada si al solicitar el cambio actual . Si v es -0, no ha sido solicitado ningún cambio y no se lanzará ningún error. Pero si v es +0, `Number` . NEGATIVE_ZERO no tendrá más su valor inmutalbe. Internamente, al redefinir una propiedad inmutbale, el nuevo valor se compara con el valor actual usando la igualdad same-value.

El método `Object.is` nos proporciona la igualdad same-value.

Cuando usar `Object.is` o el igual triple

Además de como trata `NaN`, generalmente, la única vez en la que `Object.is` posee un comportamiento especial hacia los ceros puede resultar de interés para usar ciertos esquemas de meta-programación, sobre todo en relación a los descriptores de porpiedades cuando es deseable que nuestro trabajo replique algunas de las características de `Object.defineProperty`. Si en tu situación no requiere de esto, lo mejor es evitar `Object.is` y usar `===.` (en-US)Incluso si entre tus requisitos está poseer que la comparación entre dos valores `NaN` sea verdadera, generalmente es más fácil hacer un caso especial para ello (usando el método `isNaN` que está disponible en versiones previas de ECMAScript) que calcular cómo la operaciones van afectar a los posibles signos de los valores cero en tu comparación.