UT02 Sintaxis del lenguaje JS

**Características:**

Brendan Eich desarrolla LiveScript en 10 días para Netscape para dinamizar la web.

Estandarizacion: ECMAScript o ECMA-26.

Incluyó en: JS, ActionScript, NET W3C elige JS como estándar para HTML5.

Es una marca registrada de Oracle Coporation, y es usado con licencia por los productos creados por Netscape Communications y entidades actuales, como la fundación Mozilla.

**Hilos en el desarrollo de JS:**

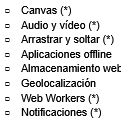
Inicios: Validación de formularios y rollovers.

DOM: Llega la animación.

Ajax: La hora de las aplicaciones web.

Bibliotecas: Hay que mejorar la productividad.

HTML5: Pilar esencial de las tecnologías revolucionaras.



(PhoneGap nos permite crear aplicaciones para dispositivos usando HTML5, CSS3 y JS).

**Características**:

Lenguaje de alto nivel **interpretado** en el lado del cliente.

Todos los navegadores actuales soportan JS.

Está **basado en objetos** con herencia por prototipos.

**Débilmente tipado.**

**Sintaxis** prestada de C y Java, pero también hereda de Awk y Perl.

Todas sus variables son globales.

Editores JS: Bloc de Notas, NotePadd++, Atom, Sublime, Brackets…

IDEs: Eclipse, VS Code, Komodo, Aptana y Netbeans.

Depurador de código: Específicos del navegador, Chrome y Firefox.

Servidor Web: NodeJS.

**Hola mundo**:

La integración de JS y HTML/XHTML podemos hacerla de 3 formas.

Directamente dentro de las etiquetas <script> y </script>.

Añadir en un fichero externo con extensión .js indicado con el atributo src.

Insertar fragmentos de JS directamente dentro de los atributos de etiquetas HTML.

Se recomienda añadir justo antes del final del </body> para que el navegador cargue primero el resto de objetos DOM y no bloquee el renderizado.

**JS en la consola**

También podemos ejecutar js directamente desde la consola del navegador (Ctrl + shift + i) o en node (ctrl + C dos veces para salir).

En node podemos ejecutar nuestro código en un archivo: clg(“”);

$ node appp.js

**Sintaxis**

**Mayúsculas y minúsculas**:

Al igual que C y Java, JS distingue entre mayúsculas y minúsculas.

Esta regla es muy importante cuando utilizamos objetos, variables, funciones o cualquier otro símbolo del lenguaje.

No es lo mismo utilizar alert() que Alert().

Se recomendaría **utilizar minúsculas** para los nombres de las variables, funciones y objetos, recurriendo a la sintaxis de camello si están compuestas por varias palabras.

**Comentarios**:

No son interpretados por el navegador.

Su función principal es facilitar la lectura del código al programador.

Una doble barra // marca el inicio de un comentario, de modo que será ignorado cualquier texto que escribamos.

Si necesitamos crear un comentario de varias líneas lo iniciaremos con /\* y \*/.

En JS puedes incluso incluir comentarios en el medio de una sentencia de código, aunque esto hace el código difícil de leer por lo que debería de utilizarse con cuidado.

**Espacios en blanco**:

Los espacios consecutivos son considerados como un único espacio.

Se consideran buenas prácticas sangrar el código utilizando como unidad de sangrado 4 espacios y no escribir líneas de más de 80 caracteres.

JS ignora los espacios, las tabulaciones y los saltos de línea con algunas excepciones.

Emplear la tabulación y los saltos de línea mejoran la presentación y legibilidad del código.

**Inserciones automáticas de punto y coma:**

Se suele insertar un signo de punto y coma (;) al final de cada instrucción de JS.

Su utilidad es separar y diferenciar cada instrucción.

Una instrucción puede extenderse a lo largo de varias líneas o podemos escribir varias instrucciones en la misma línea.

Se puede omitir si cada instrucción se encuentra en una línea independiente (o hay solo una sentencia dentro de llaves {}).

La omisión del punto y coma no es una buena práctica de programación.

**Palabras reservadas**:

Algunas palabras no se pueden utilizar para definir nombres de variables, funciones o etiquetas (class, const, enum, export…).

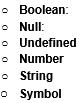
Palabras reservadas para ECMAScript 2015:



**Tipos de datos**

Los tipos de datos especifican qué tipo de valor se guardará en una determinada variable.

El último estándar ECMAScript define 6 tipos:



Existe otro tipo de datos compuesto llamado **objeto**.

JS es un lenguaje de programación de “tipado débil”.

El tipo de datos no es una propiedad de las variables, sino de los datos.

Una misma variable puede contener datos de tipo diferente en distintos instantes.

No es obligatorio declarar las variables, aunque nos permita hacerlo mediante los comandos var, let y const.

**const** se utiliza para variables inmutables.

**Cadenas de texto**:

El tipo de datos para representar cadenas de texto se llama string.

Se pueden representar letras, dígitos, signos de puntuación o cualquier otro carácter de Unicode.

La cadena de caracteres se debe definir entre comillas dobles o simples.

Para incluir comillas en una cadena de caracteres tendremos que escaparla precediéndola con el signo \ en caso de que la cadena esté delimitada por el mismo tipo (simple o doble) de comillas.

También podemos incluir cualquier carácter Unicode de 16 bits utilizando la fórmula \unnnn.

**Números**:

En JS existe sólo un tipo de dato numérico.

El valor de doble precisión de 64-bits IEEE 754.

El número comprende entre Number.MIN\_VALUE (-2^53 -1) y Number.MAX\_VALUE(2^53 -1).

No existe un tipo específico para los números enteros.

También pertenecen a este tipo los valores Infinity, -Infinity y NaN (Not A Number).

**Booleanos**:

También conocido como valor lógico.

Sólo admite 2 valores: true o false (escrito en minúsculas y sin comillas).

Es muy útil a la hora de evaluar expresiones lógicas o verificar condiciones.

**undefined**:

Este tipo de datos sólo admite el dato undefined.

Este valor se asigna implícitamente a todas las variables que han sido declaradas pero a las que no se han asignado un dato aún.

**null**:

Este tipo de datos sólo admite el dato null.

Este dato que identifica a punteros a objetos que aún no tienen un objeto asignado.

No se asigna implícitamente, sino que cuando declaremos una variable que posteriormente vamos a referir a un objeto, le asignaremos explícitamente el dato null.

**symbol:**

Introducidos en ECMAScript 6.

Es un valor primitivo único e inmutable.

Puede ser usado como la clave de una propiedad de un Object.

**object**:

Este tipo de datos puede interpretarse como una colección de otros datos, sin ninguna restricción en cuanto a su tipo, en la que cada uno de ellos está identificado por un nombre.

Para crear un dato del tipo object tenemos que utilizar la expresión new Object();

Una vez declarada la variable, podemos añadir datos a su colección utilizando la sintaxis de punto. (persona.nombre = “Pepe”;)

Los datos que contienen los objetos se clasifican en dos categorías: propiedades y métodos.

Los **métodos** son instancias de un objeto predefinido de JS llamado Function.

Todos los datos que no son de este tipo se denominan **propiedades**.

**Variables**

Se pueden definir como zonas de la memoria de un ordenador que se identifican con un nombre y en las cuales se almacenan ciertos datos.

El nombre de una variable también se conoce como identificador y debe cumplir las siguientes normas:

* Sólo pueden estar formado por letras, números y los símbolos $ y \_
* El primer carácter no puede ser un número

**Buenas prácticas de nombrado**:

Usar nombres lógicos y descriptivos para variables y funciones.

No preocuparse por la longitud de los nombres.

Usar sustantivos para nombrar las variables.

Comenzar por un verbo los nombres de funciones.

Comenzar por `is` o `has` (`es` o `tiene`) los nombres de los predicados.

Evitar nombres que no aportan información aux, temp…

En los nombres de varias palabras lowerCamelCase.

Escribir en MAYÚSCULAS los nombres de variables constantes (const) y separadas con guión bajo.

Las funciones constructoras deben comenzar por mayúsculas.

**Declaración**:

Una variable declarada con **var** dentro de una función tiene alcance dentro de esa.

ES2015 introdujo **let** y **const**.

Fuera de un bloque o función, todas las variables son globales.

Cuando JS intenta resolver una variable, busca en el ámbito de la función local.

Si no se encuentra este identificador, busca en la función externa en la que se declaró la variable, y así sucesivamente a lo largo de la cadena del ámbito hasta que alcance el ámbito global.

Si aun así no se encuentra, JS generará una excepción ReferenceError.

**Inicialización de variables**:

La asignación de datos a variables se realiza mediante el operador =, y siempre por valor.

Los datos de tipo object son la excepción, en cuyo caso la asignación se realiza por referencia.

Se pueden asignar un valor a una variable de 3 formas:

* Asignación directa de un valor concreto
* Asignación indirecta a través de un cálculo en el que se implican otras variables o const
* Asignación a través de la solicitud del valor al usuario del programa.

**Conversión de tipos**:

Los tipos de datos en JS son muy sencillos, pero a veces las operaciones no se realizan correctamente.

Es debido a la conversión de tipos de datos.

JS intenta siempre realizar la mejor conversión para realizar operaciones con datos de diferentes tipos.

A veces no es el tipo de conversión que al programador le interesa.

**Operadores**

JS es un lenguaje rico en operadores.

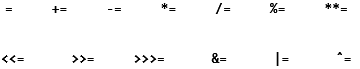
Son símbolos y palabras que realizan operaciones sobre uno o varios valores, para obtener un nuevo valor.

Cualquier valor sobre el cuál se realiza una acción (indicada por el operador), se denomina operando.

Una expresión puede contener un operando y un operador (denominado operador unario), como por ejemplo en b++, o bien dos operadores, separados por un operador (denominado operador binario), como por ejemplo en a + b.

**Asignación**:

Asigna el valor a la derecha de la expresión a la variable que está a la izquierda.



**Comparación**:

Compara los valores de 2 operandos, devolviendo un resultado de true o false (se usan extensivamente en sentencias condicionales como if… else y en instrucciones loop).

**Aritméticos**:

Unen dos operandos para producir un único valor que es el resultado de una operación aritmética u otra operación sobre ambos operandos.



**Bit a bit**:

Realizan operaciones aritméticas o de desplazamiento de columnas en las representaciones binarias de dos operandos.



**Lógicos**:

Los operadores lógicos son comúnmente utilizados con valores booleanos; estos operadores devuelven un valor booleano.

Los operadores && y || realmente devuelven el valor de uno de los operandos, así que si estos operadores son usados con valores no booleanos, podrían devolver un valor no booleano.

**Objeto**:

Ayudan a los scripts a evaluar la herencia y capacidades de un objeto particular antes de que tengamos que invocar al objeto y sus propiedades o métodos.

* nombreObjeto.pripiedad
* delete nombreObjeto/propiedad
* nombrePropiedadoNumero in nombreObjeto
* nombreObjeto instanceof tipoObjetos
* nombreObjeto new tipoObjeto

**This**

* Fuera de una función contexto global (window)
* Para obtener valores de un objeto dentro de este

**Miscelánea:**

Operadores que tienen comportamiento especial.

Operador condicional (temario) **condición** ? **valor1** : **valor2.**

Operador coma , evalúa ambos operandos y devuelve el valor del último, se usa sobre todo en bucles **for** (var i = 0, j = 9; i <= j; i++, j--).

**Typeof**: devuelve una cadena de caracteres indicando el tipo del operando evaluado.

**void (expresión)** especifica una expresión que será evaluada y no retornará ningún resultado (undefined).

**Estructuras de control**

**Sentencia compuesta**:

Una sentencia compuesta es una secuencia de sentencias y expresiones que se colocan entre llaves ({}

).

Se puede utilizar para realizar varias tareas siempre que se espere una sentencia única.

**Sentencia if … else**:

Ejecuta una sentencia si una condición especificada es evaluada como verdadera.

Si la condición es evaluada como falsa, otra sentencia puede ser ejecutada.

**Operador condicional temario**:

Similar a la función IF de Excel.

Devuelve una de las dos expresiones posibles, dependiendo de una condición.



**Sentencia switch**:

Evalúa una expresión, comparando la expresión con un conjunto de valores predefinidos, y ejecuta comandos según el caso.

**Bucle for**:

Crea un bucle que consiste en tres expresiones opcionales, encerradas en paréntesis y separadas por puntos y comas, seguidas de una sentencia ejecutada en un bucle.

**Bucle for … in**:

La sentencia for.. in itera sobre todo las **propiedades de un objeto,** en un orden arbitrario.

Para cada una de las propiedades, se ejecuta la sentencia especificada.

**Bucle for each..in**:

Es similar a for…in, pero itera sobre los **valores de las propiedades del objeto** en vez de sobre los mismos nombres de las propiedades.

**Bucle while:**

Crea un bucle que ejecuta una sentencia especificada mientras cierta condición se evalúe como verdadera.

Dicha condición es evaluada antes de ejecutar la sentencia.

**Bucle do … while**:

Crea un bucle que ejecuta una sentencia especificada, hasta que la condición de comprobación se evalúa como falsa.

La condición se evalúa después de ejecutar la sentencia, dando como resultado que la sentencia especificada se ejecute al menos una vez.

**Buenas prácticas de programación en JS**

**Consideraciones generales**:

El código debe ser mantenible.

Debemos pensar que, seguramente, no seremos los únicos que trabajaremos con él, por tanto, trataremos que nuestro código sea:

* Intuitivo
* Comprensible
* Adaptable
* Exensible
* Depurable
* Testeable

En JS Garden ofrecen información sobre los detalles escabrosos de JS.

No mezclar código JS con otros lenguajes de programación.

**Bloques**:

Usar siempre bloques explícitos.

Evitar realizar asignaciones en las sentencias condicionales **if,** **while**, etc.

No dejar abiertos los bloques **if** ni **switch**, incluir siempre **else** y default.

Usar la variable 1TBS (One True Brace) del estilo K&R.

**Sangrado y espacios**:

Cuánto menos código tengamos en una línea, menor probabilidad tendremos de tener un conflicto al mezclarlo (merge) con los cambios de otros desarrolladores.

Adaptarse al modo de sangrado (indentation) acordado en el proyecto en el que se participe. En caso de que no haya ninguno definido, está bastante extendido el uso de **cuatro espacios para el sangrado**.

Utilizar el espacio en blanco para mejorar la legibilidad del código.

Toda palabra reservada seguida de paréntesis “(“ se debe separar por un espacio.

No usar espacios en blanco en la llamada de una función entre el nombre de la función y el (. Esto ayudará a distinguir las palabras clave.

Todos los operadores binarios se deben separar de sus operandos por un espacio en blanco.

No se debe utilizar espacio entre un operador unario y su operador excepto cuando el operador es una palabra como **typeof**.

Evitar líneas más de 80 caracteres.

Si hay que romper una línea, hacerlo después de un operador, idealmente después de la coma.

**Funciones y variables**:

Evitar la declaración de variables y funciones en ámbito global, incluirlas en un espacio de nombres.

Declarar todas las variables al comienzo del bloque de código en el que se vayan a utilizar.

Cuidado con la sentencia **return** en funciones, escribir una sola, como última sentencia y terminar con ;.

No comprarar variables con **null.** Si se utiliza **instanceof** para comprobar tipos de objetos y **typeof** para tipos del lenguaje atención a cómo funcionan estos operadores.

No se recomienda el uso de **eval.** solo en casos justificados y cuando se han validado correctamente los datos a evaluar.

Las cadenas de caracteres suelen cambiarse a lo largo del desarrollo.

No escribirlas literalmente en el código, usar constantes en su lugar.

En particular urls, cadenas mostradas al usuario, ajustes, etc.

Esto es aplicable a otros tipos de datos como los númericos por ejemplo.

**Objetos**:

No modificar objetos que no se han definido.

Crear y lanzar nuestros propios objetos **Error.**

Cuidado con la función **parseint** y las bases (octal, hexadecimal…).

Utilizar arrays cuando los nombres de los miembros sean enteros secuenciales.

Utilizar objetos cuando los nombres de los miembros sean cadenas de caracteres arbitrarias.

Usar {} en lugar de **new Object().**

Usar [] en lugar de **new Array().**

**Otras:**

Usar === en lugar de == y !== en lugar de !=.

Para facilitar la legibilidad del código no se recomienda usar los operadores ++ -- ?:

Cuidado con **switch/case**, si se utilizan, se debe utilizar break.

No se recomienda el uso de las siguientes palabras reservadas: **break, continue, void, with.**

Utilizar siempre punto y coma al final de una sentencia.

Cad punto y coma en la parte de control de una sentencia for se debe seguir de un espacio en blanco.

Utilizar herramientas adecuadas para detectar problemas en el código y mejorar la calidad del mismo.

**Linting**

Es el proceso de ejecutar un programa para analizar código en busca de errores potenciales.

Un Lint o Linter es un programa que soporta linting (verificaciones de calidad del código). Están disponibles para la mayoría de lenguajes de programación y marcas: JS, CSS, HTML, Python…

Algunos de los linters más útiles son: JSLint, CSSLint, JSHint, PYlint.

Muchos editores lo tienen integrado de forma nativa como sublime, brackets, etc, o puedes instalarlo como en VS Code EsLint.

**Strict Mode**

Introducido en ES6/ECMAScript 5 es una manera opcional de escribir una variante estricta de JS.

Hace algunos cambios a la semántica normal de JS:

* Elimina algunos fallos silenciosos lanzando errores
* Arregla fallos que hacen difícil a los motores de JS realizar optimizaciones
* Prohíbe sintaxis que sean utilizadas en futuras versionaes de EMCAScript