**Desarrollo Aplicaciones Móviles**

**JavaScript**

**· Metodología.**

La metodología que vamos a seguir en este curso va a ser principalmente práctica, durante las presentaciones constantemente veréis ejemplos de todos los conceptos que vayamos explicando en código fuente que podéis y debéis probar para familiarizaros lo antes posible con el lenguaje y la tecnología.

**· Capítulo 1.- Introducción y Entorno de Desarrollo**

**·¿Qué sabemos de JavaScript?**

· ¡¡**No conocemos Javascript!!**

**· Capítulo 1.- Re-introducción y Entorno de Desarrollo**

**· ¿Quién es JavaScript?**

* Historia -> Desarrollado por Brendan Eich en 1995 en Netscape. Originalmente se llamaba Live Script, pero cuando iba a ser lanzado, a principios de 1996, se renombró al actual JavaScript, para intentar aprovechar la creciente popularidad que en ese momento estaba disfrutando Java, a pesar de que no tienen nada que ver.
* Lenguaje de Scripting -> Es el navegador quien ejecuta el código
* Lenguaje de Scripting de lado cliente más usado del mundo.

**·** **Datos de Uso**

**· Datos de Uso**

**· ¿Quién es JavaScript?**

* Sintaxis C
* Completamente OO -> Proporciona varios mecanismos de herencia e incluso patrones para la reutilización del código
* Interpretado -> No se compila, un programa intérprete lo va ejecutando según está escrito.
* Funcional híbrido -> Es un lenguaje estructurado que incorpora características de programación funcional como que las funciones sean valores primitivos y por tanto pueden usarse como variables, funciones como parámetros, funciones como retorno y primitivas funcionales (map, filter, reduce, any, all, partial)
* Débilmente tipado -> No es necesario definir el tipo de una variable ni de un parámetro. Incluso las variables puede cambiar de tipo durante su vida.
* Dinámico -> Porque permite la inyección de código al programa en tiempo de ejecución.
* Entornos cliente y servidor
* Fácil de aprender, difícil de dominar -> Inicialmente se pensó para realizar pequeñas tareas en páginas web, pero es un lenguaje con muchas capacidades y actualmente se usa en proyectos grandes. Actualmente, muchas de las características que se diseñaron para que fuera un lenguaje sencillo, complican mucho el desarrollo cuando los proyectos se complican.
* Tremendamente potente -> Ejemplos???
* Tiene varios errores -> Cuando se lanzó se hizo muy popular muy rápidamente y nunca tuvo un periodo de prueba en el que pudiera ser pulido y corregido. JavaScript es un lenguaje muy potente e imperfecto. Poco a poco se han ido adoptando convenciones y recomendaciones que si las aprendemos y aplicamos nos permitirán dominar el lenguaje y aprovechar toda su potencia.

**· La reputación de JavaScript**

**La historia del patito feo**

* Errores del propio lenguaje
* Errores de implementación en navegadores
* Errores de implementación del DOM en navegadores
* Copy-Paste de código erróneo
* Resultado: Constantes errores de ejecución que nunca se arreglaban
* Con la aparición de los estándares web y gracias a su potencia, se está convirtiendo, en El lenguaje de la web.

**· Entorno de desarrollo y herramientas**

* Obligatorio
  + Editor de textos (Vim, Textmate, Komodo, Coda, … )
  + Navegador (Firefox, Chrome, Safari, Opera, ¿Internet Explorer?, …)
  + Debugger (Firebug, Drosera, Dragonfly, IE Developer Toolbar, …)
  + Documentación: <https://developer.mozilla.org/en-US>
* Opcional
  + IDE (Aptana, WebStorm, Eclipse, ..)
* JSLint: <http://www.jslint.com>
* JSfiddle: <http://jsfiddle.net>

**· Capítulo 2.- Conceptos de programación básicos**

**· Pero antes … Debugging** -> Explicar la consola javascript de Firebug. Imprime console.\* y la salida de la última sentencia escrita.

Todo el código de los ejemplos lo podéis y debéis probar en la consola javascript.

LAB 1 +- 5 mins

**· Estructura léxica**

* Character Set :Unicode case sensitive. -> Lleva a confusión ya que HTML no lo es onclick onClick
* Comentarios -> Igual que C
* Literales -> son expresiones, se evalúan a algún valor pero no modifican el estado del programa.
* Sentencias -> normalmente no producen valor y sí modifican el estado del programa

**· Estructura léxica**

* Identificadores y palabras reservadas
  + Identificadores -> Nombres que definen variables y funciones. Deben comenzar por carácter, \_ ó $, el resto de caracteres ya pueden tener números. Importante saber que JavaScript permite Unicode en identificadores.
  + Palabras reservadas: Son palabras que no podemos usar como nombre de nuestras variables. Existen muchas. Las reconoceremos por el color en nuestro editor/consola

**· Estructura léxica**

* Opcional ; -> JavaScript interpretará los saltos de línea como un ; cuando no pueda seguir evaluando la sentencia o expresión.

Probad estos ejemlos en vuestra consola JavaScript y decidme qué ocurre en el ejemplo no esperado

Después de ver los ejemplos, siempre usaremos ;

**· Tipos Primitivos**

**· Number**

* number -> Es el único tipo. Se comporta como un double. No existen int, float, long, double, …

**· Number: Valores especiales**

* Infinity -> JavaScript no lanza errores en caso de overflow, underflow o división por cero
  + Overflow -> Cuando el número supera el rango máximo por encima o por debajo
  + División por cero
  + Ejemplo
* 0 y - 0
  + Underflow -> Ocurre cuando el resultado está más cercano a cero que el número más pequeño representable. Si ocurre por encima 0, si ocurre por debajo -0. Es un caso muy extraño que no suele hacer falta detectar.
  + Solo relevante si se usa como divisor.
* NaN -> Retornado en los casos en que el resultado no es un valor bien definido

**· Number: Precisión**

Ojo con la precisión, debido a la representación binaria de los números, los enteros son precisos hasta 15 dígitos y los de punto flotante tanto como sea posible. Este problema es común a todos los lenguajes que usan representación binaria para los números, no solo de JavaScript.

El problema con la precisión es simplemente que el sistema de representación de números, que en este caso es el sistema binario, no puede representar algunos números con exactitud. Esto no solo ocurre en el sistema binario, sino que ocurre en todos los sistemas de representación. En nuestro sistema, que es el decimal, también ocurre. No podemos, por ejemplo, representar de manera precisa 1/3, y asumimos que es 0.33. Este error además lo asumimos porque sabemos que 0.33 \* 3 no va a dar 1. Pues esto es exactamente lo que ocurre con la precisión en JavaScript.

**· String**

Poco que decir en cuanto a su definición. Es una secuencia de caracteres. Engloba al char. Un carácter es una cadena de longitud 1.

**· Undefined y null**

* undefined -> Variable predefinida que indica que un valor no ha sido inicializado. Cuando declaramos una variable, internamente, en lugar de tomar valores ‘basura’ o null, se iguala a undefined
* null -> Objeto especial que indica que ‘no hay objeto o valor’

**· Booleanos**

Conversión (casting) implícita en comparaciones

**· Detectando tipos**

Este operador sólo es relevante con tipos primitivos, para objetos siempre retorna “object”

**· Tipos proporcionados por JS**

**· Fechas y horas**

JavaScript proporciona un API para manejar fechas y horas con facilidad.

**· Fechas y hora : Ejercicio!**

Ejercicio +- 10 mins

**· El objeto global**

Objeto que se crea al inicializar el intérprete javascript y proporciona properties, funciones, constructores, objetos, etc…

Es nuestro contexto de ejecución, y nos permite utilizar sus métodos directamente, sin tener que hacer referencia al objeto global con this.

Además, almacena todas las variables globales.

**· Wrapper Objects**

* Ejemplo 1 -> Hemos visto que s es una variable del tipo primitivo string, entonces, ¿por qué tiene métodos?. Cuando intentamos referirnos a una property de algún tipo primitivo, JavaScript lo convierte en un objeto como si se hubiera creado usando el constructor new String(s). Este objeto se usa para resolver la operación y luego se descarta.
* Ejemplo 2: -> ¿Y aquí que ha pasado? En la línea 2, JavaScript ha creado el Wrapper Object para resolver la asignación de la property ‘my\_dummy\_var’ e inmediatamente lo ha descartado. En la línea 3 Javascript crea otro Wrapper Object sobre la variable s e intenta acceder a su property ‘my\_dummy\_var’, que no existe. Con este ejemplo vemos que los tipos primitivos en JavaScript se comportan como objetos cuando intentamos leer properties o cuando intentamos ejecutar métodos, pero cuando intentamos cambiarles el estado, este cambio ocurre en un objeto temporal que se descarta inmediatamente después. Lo mismo ocurre exactamente con los tipos number y boolean, pero no con null y undefined.

**· Wrapper Objects**

* Ejemplo 1: -> Aquí hemos creado un wrapper object explícitamente
* Ejemplo 2: -> JavaScript convertirá el wrapper object en su tipo primitivo cuando sea necesario (para la comparación con ==), pero si testeamos con igualdad estricta (===) vemos que no son lo mismo.
* Aquí hemos introducido un operador característico de JavaScript, que es la igualdad estricta, para comprobar si un objeto es realmente igual a algo, no sólo en valor, sino también en tipo.

**· Conversiones de tipos (CASTING)**

Implícitas: -> JavaScript cambia el tipo de las variables a los tipos que necesita para operar sin ningún problema. Lo hemos visto antes con los booleans, donde necesita un booleano, convierte lo que haya a booleano siguiendo unas reglas y lo mismo ocurre con el resto de tipos.

**· Conversiones de tipos (CASTING)**

Explícitas: -> Cuando no lo tenemos claro o queremos marcar las conversiones explícitamente para que nuestro código sea más claro.

**· Variables**

Son identificadores con nombre que almacenan valores

**· Variables: Declaración**

* Declaración e inicialización -> ¿Qué valor tomará ‘i’ en la primera línea? ¿Y ‘a’ en la última?

**· Variables: Redeclaración y omisión**

* Redeclaración -> La redeclaración solo funciona si la variable es inicializada y internamente hace una asignación.
* Omisión -> JavaScript crea la variable COMO GLOBAL. No debe usarse por ser fuente común de errores difíciles de encontrar como veremos en la parte del scope.

**· Variables: Scope**

* Global -> Variables globales, pertenecientes al objeto global. Disponibles en cualquier sitio del código
* Local -> Variables definidas dentro de una función y sus argumentos. Eclipsan las globales si tienen el mismo nombre

**· Variables: Scope**

El problema con la omisión del operador var. **->** Lo esperado sería que scope fuera “global” y que myscope no existiera, en cambio… Recordad que una inicialización sin el operador var crea variables globales y que una redefinición con asignación realiza la asignación.

**· Variables: Scope**

Función -> Esta es una característica un poco diferente de JavaScript. En los lenguajes tipo C, las variables tienen scope de bloque , en cambio, aquí están disponibles en todo el cuerpo de la función.

**· Vairables: Scope**

Hoisting (alzar, elevar) -> Las variables que se definen dentro del cuerpo de una función están declaradas desde el comienzo de ella, antes incluso de haberse definido!, pero no inicializadas! Característica JavaScript a tener muy en cuenta y que nos va a marcar el estilo de programación. En lenguajes con scope de bloque, las variables se suelen definir lo más cerca posible al punto donde son usadas, en JavaScript debemos declararlas todas al inicio de la función para evitar equivocaciones.

**· Objetos**

Es el tipo de dato fundamental en JavaScript. Son colecciones no ordenadas de properties con nombre y valor. Los objetos se comportan exactamente igual que los hashtables (Java), dictionaries (Obj-C) o arrays asociativos(Php). Pares key->value.

Son dinámicos, es decir, en cualquier momento podemos añadir o eliminar una propiedad de un objeto o clase.

**· Objetos: El ‘prototype’**

El prototype es la forma de JS de definir la clase de un objeto. Internamente funciona como otro objeto que se asocia a cada objeto de una determinada clase y del que heredan sus propiedades y métodos.

Los objetos por tanto, tendrán una serie de propiedades heredadas, que le son transferidas desde el prototype y por tanto las comparte con todos los objetos que sean de su misma clase, y además puede tener otras propiedades propias, que no tienen porqué existir en otros objetos aunque sean de la misma clase.

**· Objetos: Creación**

Literal -> La forma más sencilla de crear objetos: escribiéndolo literalmente en el código. Crea un objeto y le añade las propiedades dinámicamente. Incluso podemos crear objetos anidados.

Todos estos objetos tienen el mismo ‘prototype’, que será el de Object, que es el tipo básico.

Operador new -> Es la invocación de un constructor.

Object.create(prototype, properties) -> Recibe como parámetros el prototype (que sería algo así como decir de qué clase queremos heredar) y opcionalmente un segundo parámetro que sería una lista de properties que queremos añadir.

Con Date no se puede hacer porque está implementado de manera diferente. Comprobar que Array() devuelve un array mientras que Date() devuelve una cadena.

**· Objetos: Propiedades**

Los atributos de las propiedades determinarán su comportamiento. Si son modificables, enumerables, etc…

**· Propiedades: Ejercicio!**

* A simple vista puede parecer poco interesante la utilización de la notación de los corchetes. Además, los editores de javascript que comprueban errores y auto-completan código nunca nos ayudarán si usamos esta notación, pero vamos a poner un ejercicio para que os deis cuenta de la potencia que tiene el uso de esta notación.
* Ejercicio: +- 5 mins;

**· Propiedades: Detección.**

La solución del ejercicio anterior tiene un problema, no detectará la propiedad si es null, 0, undefined o “”. Por tanto, para detectar si un objeto tiene o no definida una propiedad debemos usar otro método.

Operador in: Detecta propiedades propias o heredadas, recordar cuando hablamos del prototype, que un objeto podía tener propiedades que se hayan añadido dinámicamente y que, por tanto, no serán comunes a todos los objetos de la misma clase.

Método hasOwnProperty(): Detecta propiedades que sean propias al objeto, sean enumerables o no.

Método propertyIsEnumerable(): Detecta propiedades que sean propias al objeto y además enumerables.

Una vez visto esto, el ejercicio anterior deberíamos haberlo escrito usando el operador in en vez de la técnica simple de acceso a propiedades

**· Propiedades: Enumeración.**

El bucle for/in es una construcción funcional del lenguaje JS, que ejecuta el cuerpo del bucle para cada propiedad enumerable del objeto. Asigna a la variable p el nombre una propiedad de o en cada iteración.

Object.keys() es un método estático de la clase Object que recibe un parámetro que es el objeto del que se van a obtener sus propiedades que cumplan la condición de ser propias y enumerables.

Object.getOwnPropertyNames() es otro método estático de la clase Object que retorna un array con las propiedades propias del objeto que recibe por parámetro, sean enumerables o no.

**· Propiedades: Comparativa**

Aquí tenemos una tabla que nos resultará útil para saber qué método de acceso o enumeración debemos utilizar dependiendo del tipo de propiedades con que vayamos a trabajar.

**· Accessor properties**

Hasta ahora siempre hemos trabajado con propiedades que representan directamente un dato, son conocidas como ‘data properties’, pero existe otra forma de declarar properties de un objeto basada en getters y setters. Éstas propiedades se conocen como ‘accessor properties’ porque cuando se utilizan, el resultado es la ejecución del método get o set correspondiente. Esto se parece mucho a los accessor methods de Java, pero con una pequeña diferencia y es que si queremos utilizarlos alrededor de una variable como se suele hacer en Java, ésta tiene que tener un nombre distinto. Podemos usar un ‘\_’ para nombrar la propiedad alrededor de la que trabajarán los accessors.

**· Propiedades: Atributos**

Sirven para determinar el comportamiento de las properties.

**· Propiedades: Manejando atributos**

Object.getOwnPropertyDescriptor()

Como su nombre indica, sólo funciona con properties propias del objeto, si queremos obtener información de las heredadas, necesitamos preguntar al objeto que las contiene a través de la cadena de prototipos. Cómo acceder a propiedades heredadas a través de la cadena de prototipos lo veremos más adelante. Esta llamada nos da un objeto al que denominamos ‘property descriptor’

Pregunta, respecto a la implementación de ésta función, ¿por qué creéis que la property hay que pasarla como cadena?.

Object.defineProperty(object, “property”, property\_descriptor)

Crea o modifica los atributos de una property

En el ejemplo, Object.keys(o) retorna un array vacío porque la propiedad no es enumerable.

Ejercicio: +- 10 mins

**· Arrays**

No tipados, al ser JS un lenguaje débilmente, por no decir no tipado, los arrays pueden contener cualquier tipo de objetos mezclados, incluso undefined como vemos en el primer ejemplo. Esta sintaxis en cambio no es correcta en la definición con el constructor, aunque siempre podríamos añadir el elemento undefined después, pero en la inicialización sólo es posible en la definición literal.

Crecen dinámicamente. Pregunta: ¿Con qué creéis que rellena los índices vacíos?

Son objetos JavaScript exactamente igual que los demás, por tanto se le pueden añadir properties.

**· Funciones**

Definición: Usando la palabra reservada function en expresiones y sentencias. Recordar que las expresiones eran lineas que no modificaban el estado de un programa y las sentencias si.

Anidadas, esta es una característica propia de JS, define una función anidada dentro de otra y sólo es visible dentro de ésta.

**· Funciones: Scope léxico**

Esta es una característica muy importante y muy usada en JavaScript, por tanto me interesa mucho que lo entendáis bien antes de seguir adelante.

Una función JS tiene acceso a todas las variables globales, a las definidas dentro de sí misma, a las que recibe como parámetros y además a las variables definidas en el mismo bloque donde ella es definida.

**· Funciones: Argumentos**

Debido a que JavaScript es un lenguaje débilente tipado, no se especifica ni se comprueba el tipo de argumentos que una función debe recibir, pero es más, tampoco se hace ninguna comprobación sobre el número de argumentos que se reciben!!

Opcionales -> Cuando una función se invoca con menos argumentos de los definidos, éstos toman valor undefined. En estos casos es bueno asignarles valores por defecto dentro de la función.

Tener en cuenta que los argumentos opcionales deben ir al final de la función, ya que no es posible omitir el primer argumento y pasar el segundo.

Argumentos extra: Cuando se invoca una función con menos argumentos de los definidos no hay problema en usarlos, pero ¿qué pasa cuando se invoca con más argumentos de los definidos? No hay forma de nombrarlos directamente.

**· Funciones: Propiedades** -> Dado que son objetos pueden tener properties, y esto puede ser útil en algunos casos, por ejemplo.

**· CLASES**

Las clases en JavaScript son bastante diferentes de las clases a las que estamos acostumbrados en lenguajes fuertemente tipados como Java o C++. Tienen algunas similitudes y podríamos simular muchas características de estos lenguajes para hacerlo más familiar, pero debemos entender que las clases en JavaScript y su herencia basada en prototipos es muy diferente que las clases en Java o C++ y su herencia basada en clases.

Clase: Conjunto de objetos que heredan propiedades del mismo objeto prototipo

Cosas importantes de este código:

* Definimos una clase y una función factory ‘range()’ para crear nuevos objetos de tipo range.
* Utilizamos una property de la función range.prototype para almacenar el objeto prototype que define la clase.
* La función range() asigna valores propios al objeto que está creando. Estos valores son propios y únicos para ese objeto y son los que definen su estado.
* Como era de esperar, todos los objetos de este tipo comparten los métodos que heredan y en ellos se usa this para acceder a las properties del objeto concreto.
* Estamos sobrescribiendo el método toString de Object.

**· Clases: Constructores**

En el ejemplo anterior hemos visto una forma de definir una clase, pero no es la mejor manera de hacerlo, ya que no define un constructor y los programadores estamos acostumbrados a crear objetos usando constructores la mayoría de las veces.

Comparando este código con el anterior, aparte de ser más corto, podemos ver:

* Hemos nombrado el constructor con mayúscula. Esta es una convención. Los constructores definen clases y los nombres de las clases estamos acostumbrados a verlos empezar con mayúscula.
* La función constructor se invoca con la keyword new.
* Hemos eliminado la sentencia return del constructor ya que en JS un constructor devuelve implícitamente el objeto this.

**· Clases: Herencia**

La herencia en JS no es fácil de entender, por tanto, hasta que no quede bien claro no pasaremos a la siguiente sección.

**· Detectando clases**

Debido a que JavaScript es un lenguaje débilmente tipado, en muchas ocasiones necesitaremos saber de qué clase es un objeto. Para ello tenemos tres técnicas. ¿Se os ocurre cómo puede ser alguna de ellas?

Mediante el constructor. -> El constructor es una property que tienen todos los prototypes y cuyo valor es un objeto Function, pero JavaScript no siempre puede asignarle el valor esperado y en ese caso debemos manejarla nosotros mismos si queremos poder usarla con éxito.

Podemos comprobar que la property constructor existe en nuestro objeto, pero no tiene el valor que esperamos. Esto es porque directamente hemos asignado la property prototype a un nuevo objeto creado literalmente y, siguiendo la cadena de prototypes para resolver la property, encuentra la de su prototype inmediatamente superior, que en este caso es el Object.

**· Modificación dinámica de clases**

Esta forma nos da una nueva posibilidad, y es que podemos añadir nuestros propios métodos o properties a las clases predefinidas String, Number, Function, etc… y esta es una característica muy potente.

Ejercicio: +- 10 mins

**· Métodos estáticos**

En JavaScript no existen métodos estáticos que podamos definir en una clase, pero para todos los que hemos trabajado con lenguajes como Java, Php, Obj-C, etc… Los métodos estáticos nos resultan de gran utilidad y JavaScript es tan potente que nos permite simularlos.

**· Capítulo 3.- Conceptos de programación avanzados**

**· Scope chain**

Ejemplo:

En JavaScript top-level la cadena consiste en un solo objeto, el objeto global. En funciones no anidadas, la cadena consta de dos objetos, el primero es el objeto que define los argumentos y las variables locales de la función, y el segundo el objeto global. En una función anidada, la cadena consta de tres o más objetos.

Esta cadena de objetos se crea de la siguiente manera: Cuando una función es definida, guarda el scope chain que existe en ese momento. Cuando la función es invocada, crea y añade un nuevo objeto a la cadena donde guarda las variables locales. El resultado de esa cadena es el scope chain para esa invocación.

En el ejemplo, el primer objeto es creado en el momento de la definición y el segundo es creado en el momento de la invocación de la función.

Ejemplo con funciones anidadas:

Este concepto de scope chain se hace realmente interesante cuando se trabaja con funciones anidadas, ya que cada vez que la función exterior es invocada, la función anidada es redefinida y como el scope chain de la función exterior puede variar entre una invocación y otra, la función anidada será, posiblemente, diferente. El código será siempre el mismo, pero el scope chain asociado será distinto.

El funcionamiento paso a paso del ejemplo sería el siguiente:

Se interpreta el código y se define el objeto global.

En cada iteración del bucle for se invoca la función f, por tanto se añade un nuevo objeto a la cadena que contiene la definición de la variable b, la variable c, que cada vez tendrá un valor diferente, y define la función s.

En la invocación de la función s se añade un nuevo objeto a la cadena con la variable d.

El código de s siempre es el mismo, pero debido a los cambios del scope chain, cada vez funcionará de una manera distinta.

Ejercicio: +- 10 mins

Este concepto es muy importante para entender las closures que veremos a continuación.

**· Closures**

Como la mayoría de los lenguajes de programación modernos, JavaScript usa scope léxico. Esto significa que las funciones son ejecutadas usando el scope existente cuando la función fue definida, no el scope que existe cuando la función es invocada.

Para poder implementar scope léxico, las funciones, cuando son definidas, almacenan en su estado interno el código de la función y una referencia al scope chain que existe en ese momento. Esta combinación de una función mas un scope es lo que se conoce en informática como closure.

Técnicamente todas las funciones JS son closures, son objetos y tienen un scope chain asociado. La mayoría de las funciones son invocadas con el mismo scope chain que había cuando fueron definidas y el concepto de la closure no es importante. Las closures son realmente interesantes cuando son invocadas bajo un scope chain diferente al que había cuando fueron definidas. Esto ocurre comúnmente cuando el retorno de una función es otra función anidada, como en el ejercicio anterior, pero en vez de invocarla directamente, se invoca un tiempo después, cuando el scope chain se ha modificado.

El uso de closures cada vez está más extendido en programación con lenguajes modernos como JS y es importante dominarla para aprovechar su potencia.

**· Closures: Private properties**

Uno uso muy frecuente de las closures es para definir propiedades privadas de nuestros objetos. En el momento en que el constructor retorna la variable my\_private ya no es accesible desde ningún sitio excepto desde la closure, por tanto es completamente privada.

**· Capítulo 4. JavaScript en Navegadores Web**

**· HTML: Estructura**

Todos los documentos HTML están estructurados de la misma manera, con cuatro partes claramente diferenciadas:

DOCTYPE: Indica al navegador que el documento es HTML

HTML: Delimita la sección donde se encuentra el código HTML

HEAD: Delimita una sección donde se definen los metadatos de la página web como el juego de caracteres, el idioma, el creador, descripción, palabras clave, etc…

También incluirá todos los recursos JS y CSS necesarios para el correcto funcionamiento y presentación del contenido

BODY: Es la sección donde se aloja el contenido de la página propiamente dicho.

**· HTML: Elementos**

Son los componentes que usaremos para definir el contenido de nuestra página web. Serán colocados, por tanto, dentro del body.

**· El navegador como sistema operativo**

En los últimos años han sufrido una gran evolución y han pasado de ser simples herramientas para visualizar páginas web a convertirse en sistemas operativos en miniatura.

**· JavaScript en…**

JavaScript es un elemento crítico en el funcionamiento de aplicaciones web. En páginas web simplemente se usa para mejorar algunos aspectos visuales, pero una página web bien diseñada debe funcionar perfectamente sin JavaScript, en cambio, las aplicaciones web son, por definición, programas JavaScript que usan los servicios que el navegador proporciona, por tanto, no se espera que funcionen sin JavaScript.

**· Incluyendo JavaScript en documentos HTML**

Los cuatro métodos: Archivo externo, inline, como event handler y en enlaces. Solo importantes los dos primeros, los otros dos simplemente para que los conozcáis, por si los veis escritos porque antiguamente se usaban mucho. El segundo metodo de anchor es la manera correcta de definir comportamiento javascript al click de un anchor

**· El objeto Window**

Es el objeto Global para los programas JavaScript que usan el navegador como contexto. Define funciones y variables globales.

**· Capítulo 5. DOM**

**· DOM: Estructura**

El DOM es el API fundamental para manipular el contenido de documentos HTML y XML. Representa el contenido de un documento HTML como un árbol de nodos que pueden representar elementos HTML o cadenas de texto.

**· DOM: Nodos**

Existen diferentes tipos de nodos dependiendo del elemento al que representan, cada uno tiene sus propiedades particulares.

Tres subtipos básicos, el documento, los elementos y elementos de texto.

**· Seleccionando elementos**

Todos los programas JavaScript que manipulan documentos HTML necesitan acceder a sus elementos

* Por el atributo id -> Cualquier elemento HTML puede tener un atributo id que además debe ser único. Si por error asignamos el mismo id a dos elementos distintos, la función seleccionará el primero que se encuentre.
  + document.getElementById(‘id’);
* Por el atributo nombre -> La mayoría de los elementos pueden tener un atributo nombre. Esto es especialmente útil en formularios, cuando se trabaja con radiobuttons o checkboxes
  + document.getElementsByName(‘name’); Devuelve un objeto NodeList, que funciona como un array inmutable.
* Por el tipo
  + document.getElementsByTagName(‘tag’)
* Por sus clases css. -> Todos los elementos HTML pueden tener un atributo class que indica su clase CSS, de esta manera podemos definir grupos de elementos que comparten las mismas propiedades visuales. Igual que se pueden asignar varias clases a un objeto, este método acepta una cadena con muchas clases separadas por espacios ‘class1 class2 class3’ y devolverá aquellos que pertenezcan a todas las clases.
  + document.getElementsByClassName(‘css\_class’)
* Por un selector css -> CSS tiene una sintaxis muy potente, conocida como selectores, que nos permiten seleccionar elementos o conjuntos de elementos dadas unas características. CSS lo veremos en más detalle más adelante, pero de momento, un ejemplo básico:
  + document.querySelector(‘h1’) y document.querySelectorAll(‘h1,h2’)
* -> Todos estos métodos los podemos aplicar a un cualquier Element, para reducir la búsqueda solamente a los nodos que sean descendientes del invocado. Cuidado con invocar selección basada en selectores css porque si el selector busca un ancestro, no lo va a encontrar.

**· Recorriendo el DOM** -> Cuando necesitamos acceder a elementos en posiciones relativas a otros elementos, por ejemplo, su padre, sus vecinos, o sus hijos, necesitamos poder movernos en el árbol que representa el documento. Para ello tenemos estas propiedades de Node:

* parentNode
* childNodes
* firstChild, lastChild
* nextSibling, previousSibling

**· Información de los nodos:** Los nodos tienen otras propiedades que nos da información de su tipo y contenido.

* nodeType: Document = 9, Element = 1, Text = 3, Comments = 8, Document-fragment = 11
* nodeValue -> El contenido textual de un nodo de texto o comentario
* nodeName -> El tag name

**· Considerando solamente los elementos** -> Para evitar los problemas que esto supone, tenemos otro juego de propiedades, solamente aplicable a elementos, no document ni nodos tipo texto.

* children
* firstElementChild, lastElementChild
* nextElementSibling, previousElementSibling
* childElementCount

**· DOM: Ejercicio**

* Ejercicio: Este api no es estándar, fue creado por IE y ha sido adoptado por la mayoría de los navegadores actuales, pero por si quisiéramos usarlo y tener la seguridad de que va a funcionar en todos, Extender la clase Element para añadir un método next

**· Atributos** -> Todos los elementos HTML pueden tener atributos asociados, vamos a ver el API del DOM para accederlos.

Los atributos son propiedades del objeto Node, por tanto podemos manejarlos igual que las propiedades de cualquier objeto JavaScript. Con notación de punto, de corchetes, podemos añadirlos y borrarlos según necesitemos. La única excepción es que no podemos borrar atributos con el operador delete, debemos utilizar el método de Node removeAttribute.

Incluso podríamos definir atributos no estándares de HTML y se añadirían al elemento HTML y éstos atributos, al no se estándares sí se podrían eliminar con el operador delete. El problema de definir atributos no estándares es que generamos HTML no válido.

**· Atributos como objetos**

La última posibilidad de trabajar con atributos de los elementos es la propiedad attributes, que devuelve un NodeList viviente con objetos de tipo Attr.

**· Manejando el contenido**

Existen varias maneras de hacer esto, pero vamos a quedarnos con una sencilla y universal.

**· Manejando el contenido**

Todos estos métodos, cuando modifican el contenido, invocan el parser del navegador, y modifican el documento con una versión parseada de la cadena. Estos métodos nos proporcionan una manera de modificar el documento usando cadenas de texto con la representación HTML, pero el DOM nos proporciona un API mucho más usable.

**· Lab 3: 15 mins**

**· Ejercicio: 10 mins**

**· Proyecto guiado: Creación de un videojuego HTML 5**

Desde ahora y hasta el final de este curso vamos a ir creando un videojuego HTML 5 muy simple. El videojuego tratará de que en la pantalla se mostrarán las letras vocales y el jugador debe colocarlas en orden para ganar.

Empezaremos por mostrar las imágenes en pantalla, aprender a moverlas, añadiremos animaciones y así poco a poco hasta completar el videojuego.

A medida que vayamos aprendiendo cosas nuevas las iremos aplicando al proyecto del videojuego así que tenéis que estar muy atentos para comprender todos los conceptos para al final poder completarlo.

**· Videojuego HTML 5**

Cargando y mostrando las imágenes

+- 15 minutos.

**· Geometría de los elementos**

Hasta ahora lo único que nos ha preocupado de los documentos HTML es su representación en árbol que hace el DOM para poder manejarlos, pero en algunos casos, como por ejemplo en nuestro videojuego, vamos a necesitar conocer exactamente la posición y el tamaño de los elementos en pantalla.

**· Capítulo 6. Eventos** -> Los programas JavaScript tienen un modelo de programación asíncrono basado en eventos. El navegador genera eventos cuando ocurre algo interesante en algún punto del documento y nosotros podemos responder ante ese evento. Para responder a los eventos tenemos que registrarlos en nuestro código, indicando qué queremos ejecutar en respuesta a qué eventos. Esto no es algo único de la programación web, todas las aplicaciones con interfaces gráficas de usuario funcionan de la misma manera.

**· Manejando eventos**

Los manejadores de eventos son propiedades de los elementos HTML, y se manejan de la misma manera que podemos manejar cualquier otro atributo, aunque con alguna particularidad.

**· Capítulo 7. CSS:** CSS es la forma estándar de definir la presentación visual de los elementos que conforman los documentos HTML. CSS está pensado para ser usado por diseñadores gráficos y maquetadores, pero también es interesante para nosotros los programadores de JavaScript porque los estilos CSS también se pueden modificar y con ello podemos conseguir animaciones. El estudio en profundidad de CSS podría suponer un módulo completo y aquí no lo vamos a ver, pero para poder modificar estilos con JavaScript tenemos que familiarizarnos con los conceptos básicos de CSS

**CSS Básico:** Existen muchas variables visuales de los elementos de un documento HTML como son fuentes, colores, espaciados, etc… CSS enumera estas variables y las define como propiedades de estilo

**Aplicando CSS:** Igual que para incluir JavaScript en el navegador tenemos varias formas, para aplicar CSS a los elementos también tenemos varias formas.

**CSS: Estilos en cascada**

CSS son las siglas de Cascading Style Sheets. Este término indica que las reglas de estilo que se aplican a cada elemento pueden venir de una “cascada” de diferentes fuentes.

Las propiedades se van acumulando y sobrescribiendo de la primera a la última y el resultado final es el estilo que se aplica a cada elemento en la página.

**· Material Extra y Webinar**

**· Capítulo 9. Frameworks: jQuery:**  A lo largo de todo el curso llevamos viendo algunos problemas comunes de JavaScript:

* Dificultad para escribir código ‘limpio’ sobre todo cuando usamos manejadores
* Incompatibilidades entre navegadores, sobre todo con IE8 y anteriores, que nos obligan a comprobar constantemente si podemos hacer uso o no de alguna característica.
* Para facilitar la escritura de código, hacerlo más legible y ligero y sobre todo, para eliminar diferencias entre navegadores, los programadores web solemos hacer uso de frameworks JavaScript.
* Existen muchos frameworks, pero jQuery es el más extendido y por eso su uso es recomendable, porque en Internet tenemos muchos recursos, además tiene muy buena documentación y sobre todo, porque es muy potente y fácil de usar.

**Iniciación a jQuery:** jQuery define una sola función global llamada jQuery(), que es tan frecuentemente usada que el Framework también define el símbolo global $ como un acceso rápido. Esto son los dos únicos símbolos globales que jQuery define. Primeros ejemplos:

* Var divs = $(“div”) -> Retorna un conjunto de cero o más elementos DOM que corresponden con el selector introducido. En este caso, todos los elementos div de la página.
* $(“p.details”).css(“background-color”, “yellow”).show(“fast”). En este caso estamos seleccionando todos los párrafos de clase ‘details’, aplicándoles un fondo amarillo y mostrándolos con una animación. Aquí podemos ver una de las principales características de jQuery, y es que nos permite el encadenamiento de llamadas porque las llamadas retornan el propio objeto jQuery.
* $(“.clicktohide”).click(function(){ $(this).slideUp("slow"); }) -> Esta llamada selecciona todos los elementos de clase clicktohide y les asigna un manejador del evento click a cada uno, que ocasionará que desaparezca con una animación lenta de tipo slideUp.

En este punto tenemos una ligera idea de lo que vamos a poder hacer con jQuery. Lo primero, vamos a necesitar dominar los selectores CSS a fondo, y vemos también que con muy poco código vamos a poder conseguir muchos y buenos resultados.

**La función jQuery():** Es la función más importante del Framework y hay varias formas de utilizarla.

* $(‘selector’): Es la forma más común. Recibe por parámetro una cadena que es un selector CSS y nos retorna el conjunto de elementos del documento que corresponden con el selector.
  + $(‘selector’, contexto): Funciona igual que la anterior, pero si le pasamos un elemento del document o un objeto jQuery, actuará solamente sobre aquellos elementos que sean descendientes del que pasamos como argumento.
* $(element|document|window): Simplemente recubre el elemento que pasamos por parámetro con un objeto jQuery, de esta manera nos permite utilizar los métodos de jQuery para manipular el elemento en vez de tener que usar métodos DOM tradicionales. Ejemplo $(document) o $(this)
* $(‘<html tag>’, atributos opcionales): jQuery crea los elementos HTML que recibe como parámetro y retorna su objeto jQuery, pero no los inserta en el document, para insertarlos veremos más adelante algunos métodos jQuery. Ejemplo $(‘<div>’). Si le pasamos los atributos, jQuery directamente se los asignará
* var img = $("<img/>", // Create a new <img> element { src:url, // with this HTML attribute,
* css: {borderWidth:5}, // this CSS style,
* click: handleClick // and this event handler. });
* $(function()}): La función que recibe por parámetro será ejecutada cuando el documento haya sido cargado y el DOM esté preparado para ser manipulado. Ejemplo:
  + $(document).ready(f)
  + $(f)
  + jQuery(f);

**Terminología jQuery:** Vamos a ver algunos términos que se usan cuando se trabaja con jQuery.

* La función jQuery: Es el valor de los símbolos jQuery y $. Es la función que, como hemos visto antes, crea objetos, registra manejadores para invocar cuando el DOM está listo y también sirve como el namespace de jQuery.
* Un objeto jQuery: Es un objeto retornado por la función jQuery. Representa un conjunto de elementos.
* Elementos seleccionados: Cuando invocamos a la función jQuery con un selector, nos retorna un objeto jQuery que representa aquellos elementos que se correspondían con el selector.
* Una función jQuery: jQuery define algunas funciones de utilidades definidas dentro del namespace jQuery, por ejemplo jQuery.each(set, funcion), que ejecuta una función para cada elemento del set. También son conocidos como métodos estáticos. Ejemplo: $.each(a, f)
* Un método jQuery: Es un método de un objeto jQuery retornado por la función jQuery. Ejemplo: $(‘a’).each(f)
* En la documentación de jQuery veremos $.each para referirse a métodos estáticos y .each para referirse a métodos jQuery.

**Queries y resultados:** Cuando hacemos una petición jQuery pasándole un selector como argumento, jQuery nos retorna un objeto jQuery que se comporta como un array. Es entonces una forma diferente de invocar la función querySelectorAll()? Es mejor, porque:

* jQuery usa querySelectorAll para resolver la petición en aquellos navegadores en los que se soporta, pero también funciona en navegadores en los que no se soporta de manera transparente para nosotros. Si usamos querySelectorAll, tendremos que escribir una función alternativa nosotros mismos para aquellos navegadores en los que no se soporta.
* jQuery nos garantiza que todos los selectores CSS3 que soporta van a funcionar en todos los navegadores, no solo en navegadores que soportan CSS3
* El objeto jQuery funciona como un array, pero es mucho más útil que el NodeList retornado por querySelectorAll() ya que sobre éste objeto jQuery podemos utilizar cualquier método jQuery con la seguridad de que va a funcionar en todos los navegadores.
* En conclusión, usar jQuery es mejor porque nos garantiza compatibilidad entre navegadores y además nos da más funcionalidad.

**Getters y Setters:** Las operaciones más usadas con jQuery son operaciones para obtener(get) o asignar(set) el valor de atributos HTML, estilos CSS, contenidos etc… A continuación algunas características de los getters y settes jQuery:

* En vez de definir un par de métodos, jQuery define un único método como getter y setter, si recibe valor por parámetro actúa como setter, sino como getter.
* Todos los setter devuelven el propio objeto, de manera que permite encadenar llamadas
* Cuando se usa como getter, el método devuelve el valor del primero elemento del objeto jQuery sobre el que se ha invocado (debemos usar map si queremos obtener un array con todos los valores). Como devuelve el elemento de la propiedad, ya no podremos continuar encadenando llamadas, así que deben usarse como último eslabón de la cadena de llamadas.
* Los setter aceptan, aparte de valores simples, objetos con propiedades, en cuyo caso asignarán todas las propiedades y también funciones, que se usarán para calcular el valor que se debe asignar.
* Todas estas características las vamos a ver aplicadas a continuación.

**Getters y setters en acción**

* Con atributos HTML attr(): Es importante notar que como los nombres de los atributos se pasan como cadenas, podemos usar el nombre del atributo como tal o bien la versión de la propiedad JavaScript (for y htmlFor, class y className)
  + $("form").attr("action"); $("#icon").attr("src", "icon.gif"); $("#banner").attr({src:"banner.gif",
  + alt:"Advertisement",
  + // Query the action attr of 1st form // Set the src attribute  
    // Set 4 attributes at once
  + width:720, height:64});  
    $("a").attr("target", "\_blank"); // Make all links load in new windows
  + $("a").attr("target", function() { // Load local links locally and load
  + if (this.host == location.host) return "\_self"
  + else return "\_blank"; // off-site links in a new window });
  + $("a").attr({target: function() {...}}); // We can also pass functions like this $("a").removeAttr("target"); // Make all links load in this window
* Con atributos CSS css() Página 550
* Con clases CSS addClass(), removeClass(), toggleClass(), hasClass(), is() Página 550
* Con valores de elementos de formularios val() página 551
* Con contenido HTML y texto text() y HTML() página 552
* Geometría página 552
* Con data de los elementos data() y removeData() página 554

**Modificando la estructura del documento**

* **Insertando y reemplazando elementos:** Si el contenido que debe insertarse es un elemento que ya existe, entonces se mueve de sitio. Si el contenido debe ser insertado varias veces porque el objeto jQuery sobre el que hacemos la llamada representa varios elementos, entonces se clona. Todos los métodos retornan el objeto jQuery sobre el que son invocados para permitir encadenar llamadas.
* $("#log").append("<br/>"+message); $("h1").prepend("§"); $("h1").before("<hr/>"); $("h1").after("<hr/>"); $("hr").replaceWith("<br/>"); $("h2").each(function() {
* var h2 = $(this);
* // Add content at end of the #log element // Add section sign at start of each <h1> // Insert a rule before each <h1>  
  // And after as well
* // Replace <hr/> elements with <br/>  
  // Replace <h2> with <h1>, keeping content
* h2.replaceWith("<h1>" + h2.html() + "</h1>"); });
* // after() and before() can be called on text nodes, as well  
  // This is another way to add a section sign at the start of each <h1> $("h1").map(function() { return this.firstChild; }).before("§");
  + Existen otros métodos que trabajan “en el sentido contrario” a éstos.
* $("<br/>+message").appendTo("#log"); // Append html to #log $(document.createTextNode("§")).prependTo("h1"); // Append text node to <h1>s

$("<hr/>").insertBefore("h1");

$("<hr/>").insertAfter("h1"); $("<br/>").replaceAll("hr");

// Insert rule before <h1>s // Insert rule after <h1>s // Replace <hr/> with <br/>

* + Y otros que “envuelven”
* // Wrap all <h1> elements with <i> elements
* $("h1").wrap(document.createElement("i")); // Produces <i><h1>...</h1></i>
* // Wrap the content of all <h1> elements. Using a string argument is easier.
* $("h1").wrapInner("<i/>"); // Produces <h1><i>...</i></h1>
* // Wrap the first paragraph in one anchor and div
* $("body>p:first").wrap("<a name='lead'><div class='first'></div></a>"); // Wrap all the other paragraphs in another div $("body>p:not(:first)").wrapAll("<div class='rest'></div>");
* **Borrando elementos**
  + Empty() elimina todos los hijos
  + Remove(‘selector opcional’) elimina los elementos seleccionados o bien los que cumplan con el selector de entre los seleccionados. También eliminará los manejadores de eventos y data asociados
  + Detach(‘selector opcional’) funciona como remove, pero no elimina manejadores ni data asociados, se usa cuando queremos eliminar los elementos temporalmente, para reinsertarlos después.
  + Unwrap() funciona al contrario que wrap(), es decir, elimina al elemento padre de los seleccionados.
* Ejercicio: Proyecto jqgallery.dom. Antes de empezar, una recomendación, para distinguir las variables cuando son objetos jQuery de cuando no, una buena técnica es nombrarlas con $ delante, por ejemplo var $variable

**Manejando eventos con jQuery:** Tiempo atrás hemos visto las dificultades que existen para manejar eventos con IE8 y anteriores, con jQuery vamos a ver que estas incompatibilidades son transparentes para nosotros.

* Manejo de eventos básico: jQuery define un método para registrar cada tipo de evento. Ejemplo:
  + $("p").click(function() { $(this).css("background-color", "gray"); });
  + jQuery asignará el manejador a cada uno de los elementos seleccionados.
* blur() focusin() change() focusout() click() keydown() dblclick() keypress() error() keyup() focus() load()
* mousedown() mouseenter() mouseleave() mousemove() mouseout() mouseover()
* mouseup() resize() scroll() select() submit() unload()
  + jQuery nos asegura que todos estos manejadores funcionarán en todos los navegadores donde lo usemos.
* El objeto evento. jQuery crea su propio objeto evento para ocultar las diferencias entre navegadores vistas hasta ahora. Todos los manejadores que creemos reciben como primer argumento un objeto evento que describe los detalles del evento. Este objeto define las funciones preventDefault(), stopPropagation() y stopImmediatePropagation().
* Manejo de eventos avanzado.
  + jQuery nos proporciona la función bind(), que nos permite hacer cosas como:
    - $('p').bind('click', f);
    - $('a').bind('mouseenter mouseleave', f);
    - $('a').bind({mouseenter:f, mouseleave:g});
  + one() registra el manejador, y una vez que se ha invocado se desregistra.
* Desregistrando manejadores: unbind()
  + $('a').unbind("mouseover mouseout");
  + $('#mybutton').unbind('click', myClickHandler);
  + $('a').unbind({ // Remove specific mouseover and mouseout handlers
    - mouseover: mouseoverHandler,
    - mouseout: mouseoutHandler });
* Lanzando eventos

**· Múltiples ventanas y frames página 371.**

* **Ejercicio mover una bola de uno a otro.**
* **Clases en ECMAScript 5 página 256 del libro**
* **Técnicas comunes de orientación a objetos en JavaScript página 233.**
* **Fast loading websites https://developer.mozilla.org/en-US/docs/HTML/Tips\_for\_authoring\_fast-loading\_HTML\_pages**
* **Fragmentación y browser testing página 347**
* **Análisis de whenready página 484**
* **Extendiendo jQuery con plugins: página 601**
* **Expresiones regulares página 269**
* **UNOBTRUSIVE JavaScript**
* **Inclusión dinámica de scripts página 338, si vale como require**
* QUEDA PENDIENTE EL CAPITULO DE EXPRESIONES Y OPERADORES, SOBRE TODO PARA OPERADORES ESPECIALES COMO VOID, DELETE, EVAL, ETC…
* QUEDA PENDIENTE EL CAPITULO DE SENTENCES PARA EL FOR..IN, LOS JUMPS, WITH, “use strict”
* **Functional Partial application(currying) con bind() en página 206**
* **Programación funcional con JavaScript en página 209**
* **Monkey-Patching ejemplo en página 205**
* COMET