



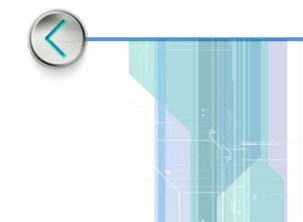
# JavaScript y DOM

**INF 467 - ICI 549**  
**Tecnologías para el Desarrollo de Aplicaciones Web**


**Prof. Enrique Urra Coloma** ([enrique.urra@gmail.com](mailto:enrique.urra@gmail.com))

Escuela de Ingeniería Informática  
Pontificia Universidad Católica de Valparaíso

## Contenidos



- ❖ **JavaScript**
- ❖ **Document Object Model**
- ❖ **Ejercicios**



Enrique Urra C. (INF 467 - ICI 549)



Tecnologías para el Desarrollo de Aplicaciones Web

# JAVASCRIPT

Enrique Urra C. (INF 467 - ICI 549)



## JavaScript

Lenguaje “**ligero**” orientado principalmente a **manipulación** de documentos Web

- Interpretado.
- Basado en prototipos.
- Tipado dinámico.
- Funciones de primera clase.
- Paradigmas múltiples.



Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## ❖ Creado por **Brendan Eich** (Mozilla Corporation) bajo el nombre "**Mocha**"

- Primera aparición en el navegador **Netscape** (1995).
- Su denominación intermedia fue "**LiveScript**".
- Adoptado por Microsoft en IE bajo el nombre de "**JScrip**t" (1996).
- Estandarizado bajo el nombre de "**ECMAScript**" (1996).

## ❖ Fue considerado por mucho tiempo un lenguaje "**amateur**", hasta la popularización de **AJAX**.

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Incorporación en HTML

- Escribiendo directo en el documento

```
<head>
  <!-- ... -->
  <script type="text/javascript">
    // Acá va el código
  </script>
  <!-- ... -->
</head>
```

- A través de un archivo externo (recomendado)

```
<head>
  <!-- ... -->
  <script type="text/javascript" src="archivo.js"></script>
  <!-- ... -->
</head>
```

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript

## Sintaxis básica (I)

```
var hola = 1;
hola = 2; // var es omitible en ciertos casos
b = 3 // No termina en ";", pero se interpreta con el salto de línea

if(hola != hola)
    alert("Entro al if!");

hola = "cadena"; alert(hola); // Ahora funciona como string

var varundefined;
alert(varundefined + " <- undefined");

// Otros tipos y estructuras
var check = true;
var pf = 0.1;

while(check)
{
    pf += 0.1;
    check = pf < 1;
}
```

Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript

## Sintaxis básica (II)

```
// Jugando con funciones...
alert(unaFuncionConParametro(10));

function unaFuncionConParametro(parametro)
{
    return "resultado! " + parametro;
}

var varQueGuardaFuncion = function(parametro) // anónima
{
    return "resultado2! " + parametro;
};

alert(varQueGuardaFuncion(5));

var otraVariableDeFuncion = varQueGuardaFuncion;
alert(otraVariableDeFuncion(10));
```

Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript

## Sintaxis básica (III)

```
// Arreglos...
var arregloA = new Array();
arregloA[0] = 1; arregloA[1] = 2; arregloA[3] = "hola";

var arregloB = new Array(true, undefined, 5, 8.5);

function listaArreglo(arreglo)
{
    for(index = 0; index < arreglo.length; index++)
        alert(arreglo[index]);
}

listaArreglo(arregloA); listaArreglo(arregloB);

var arregloC = new Array();
arregloC[1] = "casa";
arregloC["test"] = "hola";
arregloC["test2"] = undefined;
arregloC[undefined] = "test2";

for(var key in arregloC)
    alert(key + " => " + arregloC[key]);
```

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript

## Alcance de las variables (scope)

- Con la palabra reservada **var**, las variables se **declaran** en el contexto respectivo.
- Si en algún momento se utiliza un identificador **sin ser declarado**, ya sea asignando su valor u obteniéndolo, se **resuelve** automáticamente el **alcance** de la variable respectiva.
- El alcance se resuelve como una **cadena**
  - Se parte revisando el contexto más **"local"**.
  - Si no se encuentra nada en un contexto, se pasa a revisar el contexto **más general que le sigue**.
  - Esta revisión termina cuando se llega al contexto **"global"**.

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Alcance de las variables (scope)

```
var global1 = 1; // Global (para cualquier contexto)
global2 = 2; // Se auto-declara como global
global3 = global4; // Error: global 4 no se ha declarado

function funcionPadre()
{
    var local1 = 1; // Local (para funcionPadre() y contextos hijos)
    var local2 = global2; // Uso de variable global
    global5 = 5; // Se auto-declara como global

    function funcionHija() // función "interna"
    {
        var local1 = 5; // No afecta la del padre
        alert(local2); // Accede a la del padre
    }

    funcionHija();
}

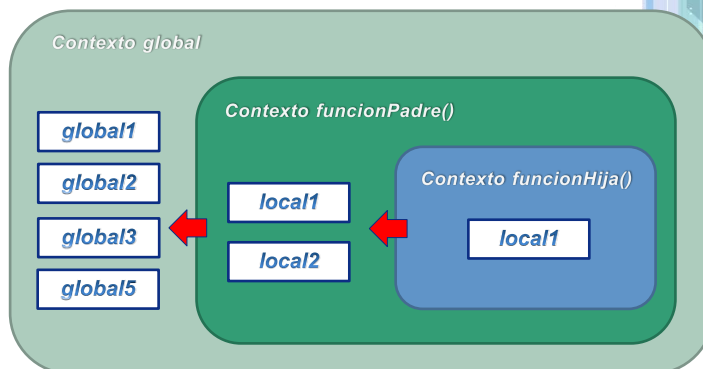
funcionPadre();
alert(local1); // Error: contexto más específico.
alert(global5); // Auto-declarada en funcionPadre()
```

Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript



## Alcance de las variables (scope)



Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript



## Closures

- El **alcance** de las variables modifica su **tiempo de vida** acorde a como han sido “referenciadas”.
- Se pueden generar **closures** para poder **acceder** a variables aún **fuera del contexto** donde ellas fueron **definidas**.



Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Ejemplo de *closure* en JavaScript:

```
function externa(paramExt) // Recibe 10 en (a)
{
    var varExterna = 0;

    function interna(paramInt) // Recibe 5 en (b)
    {
        varExterna++; // Cambia a 1 en (b)
        return paramInt + paramExt + varExterna;
    }

    return interna;
}

var guardaFunc = externa(10); // (a)
alert(guardaFunc(5)); // (b), imprime "16"
```

❖ ¿Qué pasa si se ejecuta “*guardaFunc(5)*” de nuevo?

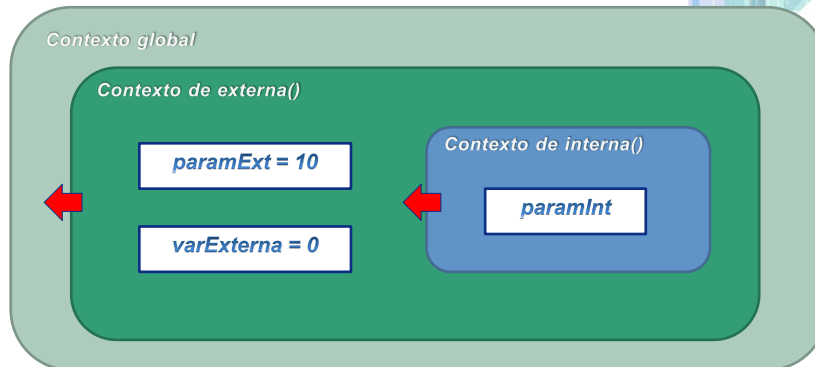
Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Closure (paso a paso)

1) Se ejecuta *externa(10)*, definiendo el contexto para *externa()* e *interna()*



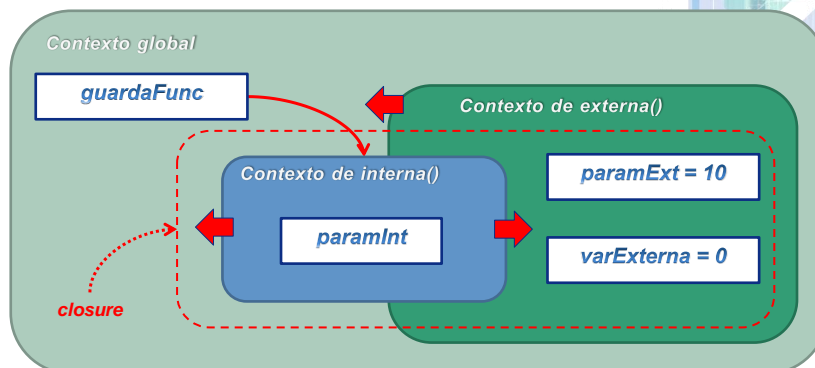
Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript



## Closure (paso a paso)

2) *externa()* retorna la referencia a *interna()* que se guarda en la variable *guardaFunc*



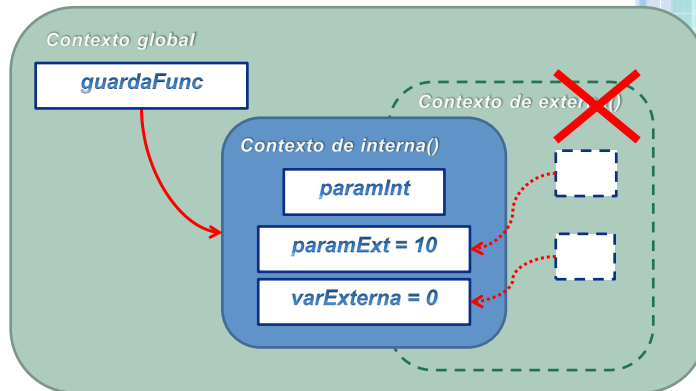
Enrique Urrea C. (INF 467 - ICI 549)



# JavaScript

## Closure (paso a paso)

3) *externa()* finaliza, algunas de sus variables siguen “siendo parte” del contexto de *interna()*



Enrique Urra C. (INF 467 - ICI 549)

# JavaScript

## Closures

- Aplicación de *closure*: patrón de módulo

```
var miAPI = (function() {  
    var variablePrivada = "algun valor";  
  
    function metodoPrivado() {  
        return variablePrivada;  
    }  
  
    return {  
        variablePublica: "Imprime",  
        metodoPublico: function() {  
            return " " + metodoPrivado();  
        }  
    }  
})();  
  
// retorna "Imprime algun valor"  
alert(miAPI.variablePublica + miAPI.metodoPublico());
```

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Orientación a Objetos (OO)

- JavaScript entrega **mecanismos** para trabajar bajo un paradigma OO, pero con algunas **particularidades** respecto a otros lenguajes
  - No **existe** el concepto de “**clase**”, en su lugar se pueden definir **funciones** que se comportan como constructores.
  - Por su naturaleza **interpretada**, los objetos **pueden cambiar** de muchas formas en tiempo de ejecución.
  - Se pueden usar **prototipos**, los cuales ayudan a cambiar los atributos y métodos de múltiples objetos a la vez.



Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript



## Orientación a Objetos (OO)

- Un objeto en JavaScript es un **diccionario**, o una **colección asociativa** de llaves y valores.

```
// Objeto sin atributos ni métodos
var objeto1 = new Object();
// Igual que arriba
var objeto2 = {};

// Setea el atributo "atr1" con valor "hola"
objeto1["atr1"] = "hola";
// Setea el atributo "atr2" con valor "10"
objeto1.atr2 = 10;

// Obtiene el atributo "atr2"
alert(objeto1.atr2);
// Obtiene el atributo "atr1"
alert(objeto1[atr1]);
// Error: No definido.
alert(objeto2.atr1);
```

Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript



## Orientación a Objetos (OO)

- Se pueden instanciar **objetos complejos** de forma **literal**

```
var objeto3 = {  
  atributo1 : "valor",  
  atributo2 : 5,  
  atributoArreglo : [1, 2, "hola", true],  
  atributoObjeto : {  
    subatributo1: "valor",  
    subatributo2: 20  
  }  
};
```

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Orientación a Objetos (OO)

- Se pueden implementar **"clases"** a través de funciones.
- Se puede **instanciar** un objeto usando el operador **"new"** e invocando una función
  - El código de la función se **ejecuta**, y en el mismo se puede realizar toda la **inicialización** del objeto. Esta función actúa como un **constructor**.
  - En el **código** de la función se puede usar una referencia a **"this"** para especificar atributos y métodos públicos.
  - También se puede usar **"var"** para atributos y métodos privados.
  - La llamada **retornará** el **objeto generado** de esta forma.
  - Los datos privados dentro del constructor permanecen en el tiempo ya que se forma un **closure** con el objeto generado.

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript

## Orientación a Objetos (OO)

### ■ Declarando la “clase”

```
// Constructor con parámetros
function MiConstructor(num1, num2)
{
    this.num1 = num1 || 0; // atributo público
    var num2 = num2 || 0; // atributo privado

    this.unMetodo = function() // Método público
    {
        alert(this.num1);
        otroMetodo();
    }

    var otroMetodo = function() // Método privado
    {
        alert(num2); // this.num2 = undefined
    }
}
```

Los métodos (funciones) declarados son **referencias**. Cada vez que se instancia un nuevo objeto “MiConstructor”, se crean **nuevas** instancias de sus **métodos**!

Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript

## Orientación a Objetos (OO)

### ■ Accediendo a los miembros de la clase

```
var instancia = new MiConstructor(2, 20);

instancia.unMetodo(); // Correcto, es público
alert(instancia.num1); // Correcto, 2

alert(instancia.otroMetodo); // undefined
alert(instancia.num2); // undefined
```

Enrique Urrea C. (INF 467 - ICI 549)

# JavaScript



## Prototipos

- Todo constructor (función) tiene una referencia a otro objeto que es su **prototipo** (*prototype*).
- Se puede acceder al prototipo con la sintaxis *NombreConstructor.prototype*.
- El prototipo guarda atributos y métodos que son **compartidos** por **todos** los objetos generados por el constructor asociado
  - Al **acceder** a algún atributo/método de un objeto, este se busca primero en la instancia misma.
  - Si **no se encuentra** en la instancia, se busca en el **prototipo** del constructor que instanció al objeto.
  - Esto emula “**herencia**” de alguna forma.

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript



## Prototipos

### ▪ Ejemplo de uso de prototipo

```
MiConstructor.prototype.metodoProto1 = function()
{
    alert(this.num1 + this.num3); // Correcto
}

MiConstructor.prototype.metodoProto2 = function()
{
    alert(num2); // Error: fuera de scope!
}

MiConstructor.prototype.num3 = 5;

var instancia2 = new MiConstructor(1, 20);
var instancia3 = new MiConstructor(2, 20);

instancia2.metodoProto1(); // 6
instancia3.metodoProto1(); // 7
instancia2.metodoProto2(); // Error
```

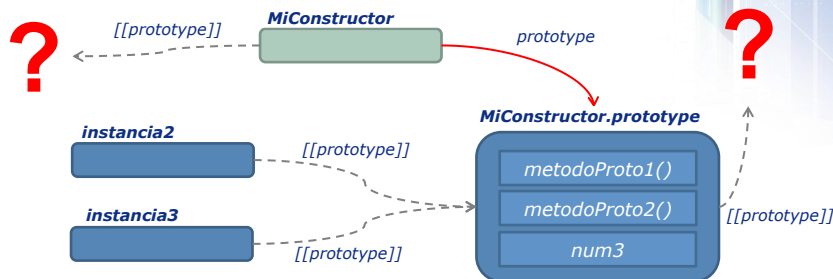
¿Se podrá **acceder** a la variable “**num2**” a través del prototipo?

Enrique Urra C. (INF 467 - ICI 549)

# JavaScript

## Prototipos

- Todo constructor (función) es un **objeto**.
- Cuando se declara un constructor, se genera su atributo **prototype** de forma automática como otro objeto.
- Todo objeto tiene además un atributo **interno e inaccesible** llamado **[[prototype]]**, al que se asigna (cuando se usa **new**) la referencia a **prototype** de la función con que se creó.
- A través de **[[prototype]]** es que se buscan (recursivamente) los atributos que **no se encuentran** en la instancia de algún objeto.

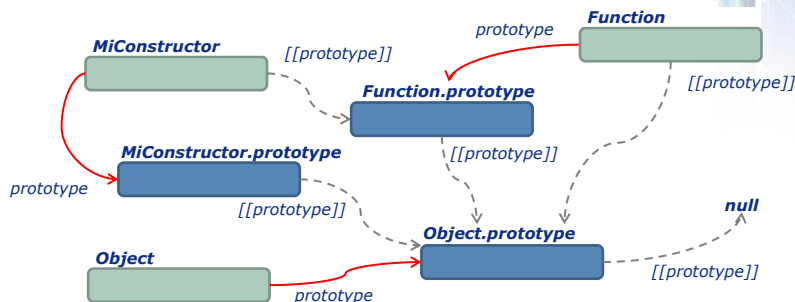


Enrique Urra C. (INF 467 - ICI 549)

# JavaScript

## Prototipos

- Para un objeto de una función, su atributo **[[prototype]]** es **distinto** a su atributo **prototype**, ya que las funciones se crean con un constructor llamado **Function()**.
- A la vez, **todo** objeto (incluido el constructor **Function()**) es generado a través del constructor **Object()**.
- Toda la **cadena** de atributos **[[prototype]]** termina efectivamente en **Object.prototype**.



Enrique Urra C. (INF 467 - ICI 549)



Tecnologías para el Desarrollo de Aplicaciones Web

# DOCUMENT OBJECT MODEL

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model

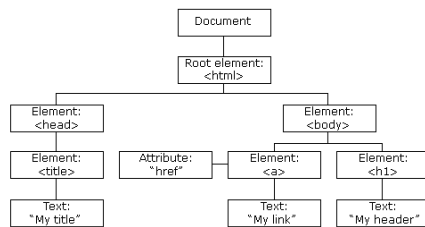


- ❖ Una de las **aplicaciones** más importantes de JavaScript es poder manipular la **estructura** y **propiedades** de documentos Web, como HTML y XML.
- ❖ Lo anterior se realiza a través del Document Object Model (**DOM**)
  - **API** “estandarizada” en los distintos navegadores conocidos.
  - Nace con la aparición de JavaScript
    - **Legacy DOM** (1996): validación de formularios.
    - **DOM intermedio** (1997): Dynamic HTML (DHTML), manipulación de CSS.
    - **DOM estandarizado** (1998 y después): Event model y soporte XML.

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model

- ❖ El DOM considera la estructura del documento como un “**árbol**” de nodos que representan elementos.
- ❖ El punto de acceso a todo este árbol, es la variable global **document**.



```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href="link">My link</a>
    <h1>My header</h1>
  </body>
</html>
```

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model

### Tipos de nodos principales

- **Document**: Nodo que es **raíz** de todo documento (en el atributo *document*).
- **Element**: Nodo asociado a **cualquier elemento** dentro del documento, al que se asocia una etiqueta.
- **Attr**: Representa un atributo (par **llave-valor**) para un elemento cualquiera.
- **Text**: Representa el contenido en **texto** de un elemento.

Enrique Urra C. (INF 467 - ICI 549)



## Document Object Model

Un nodo **cualquiera** dentro del documento es definido en JavaScript a través de un objeto de tipo **Node**

Propiedad/Método	Descripción
<b>nodeName</b>	Entrega un String que representa el nombre (tag) del nodo. Aplicable en general a todos de tipo <i>Element</i> .
<b>nodeType</b>	Entrega un entero que representa el tipo de nodo, los que estan definidos en el mismo objeto (ej: Node.ELEMENT_NODE = 1).
<b>childNodes</b>	Entrega un objeto de tipo NodeList que contiene todos los hijos del nodo actual.
<b>appendChild(node)</b>	Agrega un nodo al final del NodeList respectivo.
<b>removeChild(node)</b>	Elimina un nodo del NodeList respectivo.

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model

- ❖ Para poder **procesar** los nodos dentro de **body**, hay que asegurarse que se haya **generado** la estructura del documento adecuadamente.
- ❖ En base al **event model** del DOM, se puede configurar una función en JavaScript que se llame **justo después** de haber cargado todo el contenido de **body**.

```
<html>
  <head>
    <title>My title</title>
    <script type="text/javascript"
src="demo1.js"></script>
  </head>
  <body onload="prueba();">
    <a href="link">My link</a>
    <h1>My header</h1>
  </body>
</html>
```

El código dentro del atributo **"onload"** se llamará cuando el tag **body** se cargue completamente.

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model



### demo1.js

```
// Pruebas en Firefox
function prueba()
{
    alert(document.nodeName); // "#document"
    alert(document.childNodes.length) // 2

    var childNodes = document.body.childNodes;

    // Imprime: "#text", "A", "link", "#text", "H1", "#text"
    for(i = 0; i < childNodes.length; i++)
    {
        alert(childNodes[i].nodeName);

        if(childNodes[i].nodeName == "A")
            alert(childNodes[i].getAttribute("href"));
    }
}
```

Cuando imprime "#text" esta considerando los **nodos de texto** de los saltos de línea entre distintos tags

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model



El objeto en *document* ofrece **métodos** para **manipular** elementos de forma **transversal** en el documento

Método	Descripción
<b>getElementById(id)</b>	Retorna el nodo del elemento en el documento con el id especificado.
<b>getElementsByTagName(name)</b>	Retorna una lista de los nodos con el tag especificado.
<b>createElement(name)</b>	Genera un nodo <i>Element</i> dentro del documento con el tag especificado.
<b>createTextNode(text)</b>	Genera un nodo <i>Text</i> con el texto especificado.

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model

demo2.html

```
<html>
  <head>
    <title>My title</title>
    <script type="text/javascript" src="demo2.js"></script>
  </head>
  <body onload="prueba();">
    <a id="enlace" href="link">My link</a>
    <h1>header</h1>
    <h1>header <strong>2</strong></h1>
  </body>
</html>
```

demo2.js

```
// Pruebas en Firefox
function prueba()
{
  var nodoA = document.getElementById("enlace");
  alert(nodoA.getAttribute("href"));

  var nodosH1 = document.getElementsByTagName("h1");
  var primerNodoH1 = nodosH1[0];
  var nuevoTexto = document.createTextNode(" 1");
  var nuevoStrong = document.createElement("strong");
  nuevoStrong.appendChild(nuevoTexto);
  primerNodoH1.appendChild(nuevoStrong);
}
```

Enrique Urrea C. (INF 467 - ICI 549)

## Document Object Model

demo2.html  
(antes de *prueba()*)

[My link](#)

header

header 2

```
<html>
  <head>
    <body onload="prueba();">
      <a id="enlace" href="link">My link</a>
      <h1>header</h1>
      <h1>
        header
        <strong>2</strong>
      </h1>
    </body>
  </html>
```

demo2.html  
(después de *prueba()*)

[My link](#)

header 1

header 2

```
<html>
  <head>
    <body onload="prueba();">
      <a id="enlace" href="link">My link</a>
      <h1>
        header
        <strong> 1</strong>
      </h1>
      <h1>
        header
        <strong>2</strong>
      </h1>
    </body>
  </html>
```

Enrique Urrea C. (INF 467 - ICI 549)

## Document Object Model



### Algunos tipos de **eventos** que se pueden configurar como atributos en distintos elementos

Evento (atributo)	Descripción
<b>onclick</b>	Se ejecuta cuando se cliquea sobre el elemento.
<b>onmousedown</b>	Se ejecuta cuando se presiona el botón de click sobre el elemento.
<b>onmouseup</b>	Se ejecuta cuando se suelta el botón de click sobre el elemento.
<b>onkeydown</b>	Se ejecuta antes de onkeypress, cuando se presiona una tecla en el teclado.
<b>onkeypress</b>	Se ejecuta después de onkeydown, cuando se presiona una tecla en el teclado.
<b>onkeyup</b>	Se ejecuta cuando una tecla se deja de presionar en el teclado.
<b>onfocus</b>	Se ejecuta cuando el elemento es activado a través del mouse.

Enrique Urra C. (INF 467 - ICI 549)

## Document Object Model



demo3.html

```
<html>
  <head>
    <title>My title</title>
    <script type="text/javascript" src="demo3.js"></script>
  </head>
  <body>
    <input type="button" value="Mi bot&ocute;n" onclick="hiceClick()"/>
    <input type="text" value="texto" onfocus="darFoco()"/>
  </body>
</html>
```

demo3.js

```
// Pruebas en Firefox
function hiceClick() { alert("Hice click!"); }
function darFoco() { alert("Tiene foco!"); }
```

Enrique Urra C. (INF 467 - ICI 549)



Tecnologías para el Desarrollo de Aplicaciones Web

## EJERCICIOS

Enrique Urra C. (INF 467 - ICI 549)

## Ejercicios



### JavaScript (sin DOM)

- Experimentar con los ejemplos entregados en este material.
- Usar algún debugger de Javascript (como el complemento Firebug de Firefox) para los ejemplos.
- Crear y probar alguna API usando el patrón de módulo y closures.
- Implementar alguna estructura de datos conocida (como pila, cola, lista enlazada) usando objetos en JavaScript (¡respetar el encapsulamiento!).



Enrique Urra C. (INF 467 - ICI 549)

## Ejercicios



### JavaScript (con DOM)

- Experimentar con los ejemplos entregados en este material.
- Manipular la estructura de algún HTML simple a través del DOM y JavaScript
  - Revisar la W3Schools para una referencia completa: <http://www.w3schools.com/dom/>.
- Probar distintos eventos y su funcionamiento en un documento
  - Revisar la W3Schools para una referencia completa: [http://www.w3schools.com/jsref/dom\\_obj\\_event.asp](http://www.w3schools.com/jsref/dom_obj_event.asp).



Enrique Urra C. (INF 467 - ICI 549)

## Ejercicios



### Estudio individual

- Manipulación de CSS vía DOM.
- Manipulación dinámica de eventos a través del DOM.
- Compatibilidad de navegadores y DOM.



Enrique Urra C. (INF 467 - ICI 549)