Laboratorio #4: JavaScript

JavaScript

JavaScript es un lenguaje de scripting interpretado, basado en objetos, débilmente tipado, con funciones de primera clase[1].

JavaScript es el lenguaje de scripting de mayor uso en el Web. Así como el lenguaje de marcado HTML representa la estructura y el lenguaje de estilos CSS es la presentación, el lenguaje de scripting JavaScript representa comportamiento en el documento HTML.

El lenguaje JavaScript puede funcionar tanto como un lenguaje procedural como un lenguaje orientado a objeto. Los objetos son creados programáticamente en JavaScript, al añadir métodos y atributos a objetos vacíos en tiempo de ejecución, a diferencia de las definiciones de clase sintácticas comunes en lenguajes compilados como C++ y Java. Una vez un objeto ha sido construido puede ser usado como un prototipo para crear objetos similares.

El estándar de JavaScript es ECMAScript[2], implementado en los distintos navegadores. Para el año 2012, los navegadores modernos soportan ECMAScript versión 5.1. Los navegadores más antiguos soportan al menos ECMAScript versión 3. La sexta versión de este lenguaje se encuentra en desarrollo.

El estándar ECMA establece que ECMAScript es un lenguaje de programación orientado a objeto para realizar computación y manipular objetos computacionales en un ambiente anfitrión. ECMAScript no procura ser computacionalmente auto suficiente, sin embargo, se espera que el ambiente de ejecución de un programa ECMAScript provea los objetos y otras facilidades específicas al ambiente.

ECMAScript fue diseñado originalmente como lenguaje de scripting en el ambiente Web, sin embargo, ECMAScript puede proveer capacidades de scripting para una variedad de ambientes anfitriones.

Por ejemplo, JavaScript es usado en ambientes diferentes como: base de datos NoSQL como Apache CouchDB[3], en aplicaciones de red escalables como node.js[4], como lenguaje para query en el uso de MapReduce[5] para manejo de grandes datos o como lenguaje de desarrollo de para aplicaciones ETL[6].

---

[1] http://en.wikipedia.org/wiki/First-class_function; http://rosettacode.org/wiki/First-class_functions.

[2] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[3] http://couchdb.apache.org/

[4] http://nodejs.org/

[5] http://en.wikipedia.org/wiki/MapReduce

[6] http://en.wikipedia.org/wiki/Extract,_transform,_load

Un navegador web provee un ambiente anfitrión ECMAScript que incluye, entre otros, objetos que representan ventanas, menus, pop-ups, cajas de diálogo, áreas de texto, enlaces, frames, historial, cookies y entrada/salida. El navegador web como ambiente anfitrión también provee mecanismos para relacionar el lenguaje de scripting con eventos como cambio de foco, carga de documentos e imágenes, selecciones, envío de formularios y acciones del ratón. En este ambiente anfitrión el lenguaje JavaScript es reacciona a la interacción del usuario y no es necesario un programa principal.

Entre las características más relevantes de Javascript que proveen a este de gran versatilidad y facilidad de manejo.

1. **Imperativo y estructurado:** Soporta muchas de las características de los lenguajes estructurados como C, con la excepción del ámbito de las variables, dado que JavaScript utiliza ámbito a nivel de funciones.
2. **Dinámico**
    a. *Tipado dinámico:* Como en la mayoría de los lenguajes de scripting el tipo está asociado al valor en lugar de la variable, lo cual hace sumamente versátil el manejo de los elementos.
    b. *Orientado a objetos:* JavaScript es basado en objetos mediante el esquema de prototipos. Los mismos son arreglos asociativos con la posibilidad de escribirse con el estándar común de los lenguajes orientados a objetos clásicos (x.attr) bajo la forma de azúcar sintáctica. JavaScript tiene definidos un cierto grupo de objetos como function o date.
    c. *Evaluación en tiempo de ejecución:* JavaScript utiliza la función eval para ejecutar sentencias provistas en strings en tiempo de ejecución.
3. **Funcional / Funciones de primera clase:** Las funciones en JavaScript son de primera clase, son objetos en sí mismos. Por tanto ellas mismas tienen métodos. Una función anidada es una definida dentro de otra función y es creada cada vez que la función es invocada. Además, cada función creada forma una clausura léxica: El ámbito léxico de de la función externa, incluidas cualquier constantes, variables locales y argumentos se vuelven parte del estado interno de cada función interna, inclusive luego de la ejecución o cuando la función externa concluye.
4. **Basado en prototipos:** JavaScript usa prototipos en vez de clases para el uso de herencia. Es posible llegar a emular muchas de las características que proporcionan las clases en lenguajes orientados a objetos tradicionales por medio de prototipos en JavaScript.
    a. *Funciones como constructores de objetos*: Las funciones también se comportan como constructores. Prefijar una llamada a la función con la palabra clave new crea una nueva instancia de un prototipo, que heredan propiedades y métodos del constructor (incluidas las propiedades del prototipo de Object). ECMAScript 5

ofrece el método Object.create, permitiendo la creación explícita de una instancia sin tener que heredar automáticamente del prototipo de Object (en entornos antiguos puede aparecer el prototipo del objeto creado como null). La propiedad prototype del constructor determina el objeto usado para el prototipo interno de los nuevos objetos creados. Se pueden añadir nuevos métodos modificando el prototipo del objeto usado como constructor. Constructores predefinidos en JavaScript, como Array u Object, también tienen prototipos que pueden ser modificados. Aunque esto sea posible se considera una mala práctica modificar el prototipo de Object ya que la mayoría de los objetos en Javascript heredan los métodos y propiedades del objeto prototype, objetos los cuales pueden esperar que estos no hayan sido modificados.

5. **Funciones variádicas:** Un número indefinido de parámetros pueden ser pasados a la función. La función puede acceder a ellos a través de los parámetros o también a través del objeto local arguments.
6. **Expresiones regulares:** JavaScript también soporta expresiones regulares de una manera similar a Perl, que proporcionan una sintaxis concisa y poderosa para la manipulación de texto que es más sofisticado que las funciones incorporadas a los objetos de tipo string.

Sintaxis básica

Tal como fue indicado, la sintaxis de JavaScript es muy semejante a la de C (y Java) sin embargo cubriremos algunos detalles de la misma, simplemente con el uso de ejemplos.

Declaración de variables

```
var x; //Definición sin valor
var y = 2; //Definición y asignación de valor 2
```

Declaración de funciones (generales)

```
function sumar(a,b) {
        return a+b;
}
```

Función recursiva

```
function factorial(n) {
        if ( n === 0 ) {
                return 1;
        }
        return n*factorial(n-1);
```

```
}
```

Función anónima

```
var sumar =  function (a,b) {
        return a+b;
}
```


Objeto (Forma literal)

```
var persona =  {
        nombre: “Brendan Eich”,
        cedula: 111111,
        edad: 52;
        envejecer: function() { this.edad = this.edad + 1; }
};
```

Objeto (Forma dinámica)

```
var persona =  {};
persona.nombre: “Brendan Eich”,
persona.cedula: 111111,
persona.edad: 52;
persona.envejecer = function() { this.edad = this.edad + 1; };
```

Objetos como arreglos asociativos

```
var persona =  {
        nombre: “Brendan Eich”,
        cedula: 111111,
        edad: 52;
        envejecer: function() { this.edad = this.edad + 1; }
};

persona[‘nombre’]; // “Brendan Eich”
```

Herencia (Prototipo):

```
var persona =  {
        nombre: “Brendan Eich”,
        cedula: 111111,
        edad: 52;
        envejecer: function() { this.edad = this.edad + 1; }
};
var estudiante = Object.create(persona);
```

```
estudiante.carnet = 123456;
```

JavaScript tiene una característica particular respecto a los lenguajes clásicos y es que no provee una forma de realizar entrada/salida de datos. Esto es debido a que el estándar de JavaScript (ECMAScript) plantea que el entorno en el cual se ejecute, debe proveer los mecanismos de entrada/salida.

DOM

JavaScript, ambientado en el entorno web tiene una fuerte interacción con la API DOM lo cual es sumamente útil para los objetivos de comportamiento que buscamos del lenguaje y su interacción con HTML. Adicionalmente a los métodos de navegación por el árbol DOM que fueron mencionados y manejados en el laboratorio 2 en JavaScript se definen otros métodos de selección, agregación y manejo del árbol DOM.

```
document.getElementById('nombre_id'); //Obtiene el elemento con el nombre de id especificado
document.getElementsByClassName('nombre_clase'); //Obtiene LOS elementos que tengan en el atributo clase el nombre de clase indicado y devuelve un arreglo con todos ellos
document.getElementsByName('nombre'); //Obtiene los elementos con el valor nombre indicado en el atributo name y devuelve un arreglo con todos ellos
document.getElementsByTagName('nombre_etiqueta'); //Obtiene los elementos del tipo indicado y los devuelve en un arreglo con todos ellos
element.innerHTML = "<html_en_string>"; //Permite colocar html o texto como hijo del elemento
element.style.styleProperty = "<valor equivalente del CSS>"; //Permite asignar a un elemento cierto valor de una propiedad CSS
element.attribute = "valor"; //Permite cambiar o asignar el valor del atributo de un elemento
```

Igualmente en JavaScript se proveen de otro tipo de vías para manipular elementos del árbol DOM bajo la forma de métodos (.focus(), .checked(), .selectedIndex) y objetos predefinidos (.options) que facilitan el trabajo y la navegación. Queda de parte del estudiante verificar a medida que se manejen con JavaScript todos ellos.

Eventos

En el lenguaje JavaScript se ubica el comportamiento de los documentos HTML y en la API DOM se maneja el concepto de eventos. El desarrollador tiene a su disposición JavaScript para escuchar eventos del DOM y generar una respuesta que puede involucrar modificar el DOM de alguna forma.

Los eventos en el árbol DOM involucran en su gran mayoría una acción por parte del usuario, y comprenderlos y conocerlos queda de parte de la comprensión del estudiante. No obstante, navegaremos las distintas formas de manejar la asignación de acciones ante los eventos. En general existen tres niveles de manejo y asignación de los eventos descritos en los siguientes ejemplos:

```
//En HTML
<elemento onevent=”//código Javascript”>… <!-- Nivel 0 inline -->
//En JavaScript
element.onevent=<objeto_función>; //Nivel 0 tradicional
element.addEventListener(‘event’,<objeto_función>); //Nivel 2
```


Inclusión de JavaScript en HTML

Para agregar código JavaScript en HTML se proveen en general de tres formas de hacerlo.

Archivo externo

```
<script type=”text/javascript” src=”ruta/al/archivo.js”/>
```

Código explícito dentro del HTML

```
<script type=”text/javascript”>
//Código JavaScript
</script>
```

Código en un event handler de nivel 0 inline

```
<element onevent=”//Código JavaScript”>
```

Buenas prácticas

Existen ciertas ejecuciones de JavaScript que son altas consumidoras de recursos y las cuales deben ser evitadas para reducir los tiempos de ejecución.

1. **Evitar usar eval y el constructor Function:** Usar eval o el constructor de función son operaciones costosas dado que cada vez que son llamados el motor de scripts debe convertir código fuente en ejecutable.

Lento:

```
function addMethod(object, property, code) {
    object[property] = new Function(code);
}
addMethod(myObj, 'methodName', 'this.localVar=foo');
```

Rápido:

```
function addMethod(object, property, func) {
    object[property] = func;
}
addMethod(myObj, 'methodName', function () { 'this.localVar=foo'; });
```

2. **No usar try-catch-finally dentro de funciones críticas en rendimiento de ejecución**

Este elemento crea una nueva variable para el ámbito actual cada vez que la cláusula catch es ejecutada y una excepción es capturada

Lento:

```
Slow:
var object = ['foo', 'bar'], i;
for (i = 0; i < object.length; i++) {
   try {
      // do something that throws an exception
   } catch (e) {
      // handle exception
   }
}
```

Rápido:

```
var object = ['foo', 'bar'], i;
try {
    for (i = 0; i < object.length; i++) {
        // do something
    }
} catch (e) {
    // handle exception
}
```

3. **Evitar utilizar variables globales**

Si se referencian variables globales desde el ámbito de una función el motor de scripting tiene que buscar en el ámbito para encontrarlas.

Lento:

```
var i,
    str = '';
function globalScope() {
    for (i=0; i < 100; i++) {
        str += i; // here we reference i and str in global scope which is slow
    }
}
globalScope();
```

Rápido:

```
function localScope() {
    var i,
        str = '';
    for (i=0; i < 100; i++) {
        str += i; // i and str in local scope which is faster
    }
}
localScope();
```


4. **Evitar for/in dentro de funciones críticas en rendimiento de ejecución**

El bucle for/in requiere que el motor de scripting construya una lista de propiedades numerables y chequee duplicados antes de comenzar.

Lento:

```
var sum = 0;
for (var i in arr) {
   sum += arr[i];
}
```

Rápido:

```
var sum = 0;
for (var i = 0, len = arr.length; i < len; i++) {
   sum += arr[i];
}
```

5. **Evitar el uso del operador concatenado de strings**

El operador + crea un nuevo string en memoria y el valor concatenado le es asignado, luego de esto, el resultado es asignado a la variable.

Lento:

```
a += 'x' + 'y';
```

Rápido:

```
a += 'x';
a += 'y';
```

6. **Operaciones primitivas son más rápidas que llamadas a funciones**

Es mejor considerar alternativas en la forma de operaciones primitivas que realizar llamadas a funciones

Lento:

```
var min = Math.min(a, b);
arr.push(val);
```


Rápido:

```
var min = a < b ? a : b;
arr[arr.length] = val;
```

7. **Evitar referencias DOM innecesarias en objetos**

Lento:

```
var car = new Object();
car.color = "red";
car.type = "sedan"
```

Rápido:

```
var car = {
 color : "red";
 type : "sedan"
}
```

8. **Maximizar la velocidad de resolución de objeto y minimizar la cadena de ámbito**

Lento:

```
var url = location.href;
```

Rápido:

```
var url = window.location.href;
```

9. **Minimizar los comentarios y evitar nombres de variables largos**

Mantener los comentarios en el script al mínimo o evitarlos, especialmente dentro de funciones, ciclos y arreglos. Los comentarios aumentan innecesariamente la ejecución y aumentan el tamaño del archivo.

Ineficiente:

```
function someFunction()
{
var person_full_name="somename"; /* stores the full name*/

}
```

Eficiente:

```
function someFunction()
{
var name="somename";
}
```


10. Utilizar la propiedad innerHTML en lugar de operaciones del DOM

La propiedad innerHTML es más rápida de manejar que realizar operaciones DOM de creación, colocación y eliminación de elementos DOM.

Una buena práctica es implementar plantillas HTML e incluirlas mediante innerHTML.

11. Usar cloneNode()

Si no se está trabajando con elementos que no contengan elementos de formulario o manejadores de eventos es mejor clonar el elemento, modificarlo, y colocarlo nuevamente. Esto con el objetivo de evitar re-pintado de la vista.

```
var orig = document.getElementById('container'),
    clone = orig.cloneNode(true),
    list = ['foo', 'bar', 'baz'],
    elem,
    contents;
clone.setAttribute('width', '50%');
for (var i = 0; i < list.length; i++) {
    elem = document.createElement('div');
    content = document.createTextNode(list[i]);
    elem.appendChild(content);
    clone.appendChild(elem);
}
original.parentNode.replaceChild(clone, original);
```

12. Modificar un elemento invisible

Si la propiedad display de un elemento está asignada a none no será re-pintado.

Lento:

```
var subElem = document.createElement('div'),
    elem = document.getElementById('animated');
elem.appendChild(subElem);
elem.style.width = '320px';
```

Rápido:

```
var subElem = document.createElement('div'),
    elem = document.getElementById('animated');
elem.style.display = 'none';
elem.appendChild(subElem);
elem.style.width = '320px';
elem.style.display = 'block';
```

13. Realizar múltiples cambios a los estilos predefinidos mediante el uso de clasName.

14. Realizar caching de valores DOM en variables

Lento:

```
document.getElementById('elem').propertyOne = 'value of first property';
document.getElementById('elem').propertyTwo = 'value of second property';
document.getElementById('elem').propertyThree = 'value of third property';
```

Rápido:

```
var elem = document.getElementById('elem').propertyOne = 'value of first property';
elem.propertyTwo = 'value of second property';
elem.propertyThree = 'value of third property'
```


15. Remover referencias a documentos que ya han sido cerrados

Lento:

```
var frame = parent.frames['frameId'].document,
    container = frame.getElementById('contentId'),
    content = frame.createElement('div');
content.appendChild(frame.createTextNode('Some content'));
container.appendChild(content);
```

Rápido:

```
var frame = parent.frames['frameId'].document,
    container = frame.getElementById('contentId'),
    content = frame.createElement('div');
content.appendChild(frame.createTextNode('Some content'));
container.appendChild(content);
// nullify references to frame
frame = null;
container = null;
content = null;
```

16. Usar delegación de eventos

Agregar manejadores de eventos puede ser costoso si se agregan muchos elementos a los cuales se le debe agregar un manejador de eventos a cada uno.

Si por el contrario se utiliza un manejo de eventos en el cual todos los eventos se propagan al objeto del documento que esté más alto en la jerarquía. Esto significa que uno puede enlazar eventos al documento el cual invoca controladoras y pasa el objeto evento a este. El controlador es responsable por inspeccionar los detalles y despachar la lógica apropiada.

Lento:

```
var elems = [first, ..., last];
for (var i, l = elems.length; i++; i < l) {
    elems[i].onclick = function() {};
    elems[i].onblur = function() {};
 }
```

Rápido:

```
//HTML
<button id="doSomething">Click me to do something</button>

// JS
document.addEventListener('click', function(event) { eventController(event); }, false);
document.addEventListener('blue', function(event) { eventController(event); }, false);

function eventController(event) {
   // inspect the event object internals and do something wise
   if (event.target.id === 'doSomething') {
      doSomething();
   }
}

function doSomething() {}
```


17. Acelerar manejadores de eventos que se disparan excesivamente

Si un manejador es llamado muchas veces, la capacidad de respuesta de la interfaz de usuario disminuye y sobrecarga el CPU. Un buen caso es el evento resize

Lento:

```
window.onresize = resizeHandler; // fires excessively during resize
```

Rápido:

```
function SomeObject() {
     var self = this;
     this.lastExecThrottle = 500; // limit to one call every "n" msec
     this.lastExec = new Date();
     this.timer = null;
     this.resizeHandler = function() {
         var d = new Date();
         if (d-self.lastExec < self.lastExecThrottle) {
             // This function has been called "too soon," before the allowed "rate" of twice per second
             // Set (or reset) timer so the throttled handler execution happens "n" msec from now instead
             if (self.timer) {
                 window.clearTimeout(self.timer);
             }
             self.timer = window.setTimeout(self.resizeHandler, self.lastExecThrottle);
             return false; // exit
         }
         self.lastExec = d; // update "last exec" time
         // At this point, actual handler code can be called (update positions, resize elements etc.)
         // self.callResizeHandlerFunctions();
     }
}

var someObject = new SomeObject();
window.onresize = someObject.resizeHandler;
```