Programación Web
Tema 3.3 Java Script Avanzado

Miguel Ángel Manso
Emerson Castañeda
Ramón Alcarria
ETSI en Topografía, Geodesia y Cartografía - UPM

JavaScript Avanzado (I)

- **Conceptos sobre Objetos**
  - En **JavaScript** los objetos se emplean para organizar el código fuente de una forma más clara y para encapsular métodos y funciones comunes
  - La forma más sencilla de crear un objeto es mediante la palabra reservada **new** seguida del nombre de la clase que se quiere instanciar

JavaScript Avanzado (II)

```
var elObjeto = new Object();
var laCadena = new String();
```

- Por otra parte, la variable ***elObjeto*** almacena un objeto genérico de **JavaScript**, al que se pueden añadir propiedades y métodos propios para definir su comportamiento
- El objeto ***laCadena*** creado mediante la clase nativa ***String*** permite almacenar una cadena de texto y aprovechar todas las herramientas y utilidades que proporciona ***JavaScript*** para su manejo
- Como los objetos son en realidad ***arrays asociativos*** que almacenan sus **propiedades** y **métodos**, la forma más directa para definir esas propiedades y métodos es la notación de puntos, ó la notación tradicional de los **arrays**

```
elObjeto.id = "10";
elObjeto.nombre = "Objeto de prueba";
```

```
elObjeto['id'] = "10";
elObjeto['nombre'] = "Objeto de prueba";
```


JavaScript Avanzado (IV)

- A diferencia con otros lenguajes de programación, orientados a objetos, para asignar el valor de una **propiedad** **no es necesario que la clase tenga definida previamente esa propiedad**
- Tanto propiedades como métodos usan la notación de puntos

```
elObjeto.muestraId = function() {
    alert("El ID del objeto es " + this.id);
}
```

- Los métodos se definen asignando funciones al objeto. Se puede crear una función anónima y asignarla al objeto como método, si dicha función no está definida previamente, como se ve en el ejemplo anterior

JavaScript Avanzado (V)

- Un aspecto importante es el uso de la palabra reservada **this**
- La palabra **this** se suele utilizar habitualmente dentro de los métodos de un objeto y siempre hace referencia al objeto que está llamado a ese método

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.muestraId = function() {
    alert("El ID del objeto es " + this.id);
}
```

- Dentro del método, **this** apunta al objeto que llama a ese método. En este caso, **this** hace referencia a **elObjeto**
- **this** es muy utilizado. El motivo es que nunca se puede suponer el nombre que tendrá la variable (en este caso *elObjeto*) que incluye ese método

JavaScript Avanzado (VI)

- La palabra **this** se debe utilizar siempre que se quiera acceder a una propiedad de un objeto, ya que en otro caso, no se está accediendo correctamente a la propiedad

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.muestraId = function() {
    alert("El ID del objeto es "+ id);
}
```

- Si se ejecuta el ejemplo anterior, se muestra el error ***"id is not defined"*** (la variable id no está definida)
- También se puede asignar a los métodos de un objeto funciones definidas con anterioridad (además de las funciones anónimas)

```
function obtieneId() {
    return this.id;
}
elObjeto.obtieneId = obtieneId;
```

JavaScript Avanzado (VII)

- A continuación se muestra un objeto completo formado por varias propiedades y métodos (creado con la notación de puntos):

```
var elObjeto = new Object();
elObjeto.id = "10";
elObjeto.nombre = "Objeto de prueba";
elObjeto.muestraId = function() {
    alert("El ID del objeto es "+ this.id);
}
elObjeto.muestraNombre = function() {
    alert(this.nombre);
}
```

- Siguiendo este mismo procedimiento, es posible crear objetos complejos que contengan otros objetos

```
var Aplicacion = new Object();
Aplicacion.Modulos = new Array();
Aplicacion.Modulos[0] = new Object();
Aplicacion.Modulos[0].titulo = "Lector RSS";
var inicial = new Object();
inicial.estado = 1;
inicial.publico = 0;
inicial.nombre = "Modulo_RSS";
inicial.datos = new Object();
Aplicacion.Modulos[0].objetoInicial = inicial;
```

En el ejemplo anterior, se define un objeto principal llamado **Aplicacion** que a su vez contiene varios objetos
La propiedad **Modulos** del objeto **Aplicacion** es un **array** en el que cada elemento es un objeto que representa a un módulo
A su vez, cada objeto **Modulo** tiene una propiedad llamada **titulo** y otra llamada **objetoInicial** que también es un objeto con las propiedades del módulo: **estado, publico, nombre y datos** (también un objeto)

JavaScript Avanzado (XIII)

- **JavaScript** utiliza funciones para simular los constructores de objetos, por lo que estas funciones se denominan ***"funciones constructoras"***
- *El siguiente ejemplo crea una función llamada Factura que se utiliza para crear objetos que representan una factura*

```
function Factura(idFactura, idCliente) {
    this.idFactura = idFactura;
    this.idCliente = idCliente;
}
```

- Pasar al constructor de la clase una serie de valores para inicializar algunas propiedades es un concepto que también se utiliza en **JavaSript**, aunque su realización es diferente. En este caso, la función constructora inicializa las propiedades de cada objeto mediante el uso de la palabra reservada **this**

JavaScript Avanzado (XIV)

- La función constructora puede definir todos los parámetros que necesita para construir los nuevos objetos y posteriormente utilizar esos parámetros para la inicialización de las propiedades
- Normalmente, las funciones constructoras no devuelven ningún valor y se limitan a definir las propiedades y los métodos del nuevo objeto
- Después de definir la función anterior, es posible crear un objeto de tipo Factura y simular el funcionamiento de un constructor

```
var laFactura = new Factura(3, 7);
alert("cliente = " + laFactura.idCliente + ", factura = " + laFactura.idFactura);
```


JavaScript Avanzado (XV)

- Las funciones constructoras no solamente pueden establecer las propiedades del objeto, sino que también pueden definir sus métodos. Se puede crear un objeto completo llamado Factura con sus propiedades y métodos

```
function Factura(idFactura, idCliente) {
    this.idFactura = idFactura;
    this.idCliente = idCliente;
    this.muestraCliente = function() {
        alert(this.idCliente);
    }
    this.muestraId = function() {
        alert(this.idFactura);
    }
}
```

JavaScript Avanzado (XVI)

- Una vez definida la ***pseudoclase*** mediante la función constructora, ya es posible crear objetos de ese tipo. En el siguiente ejemplo se crean dos objetos diferentes y se emplean sus métodos

```
var laFactura = new Factura(3, 7);
laFactura.muestraCliente();
var otraFactura = new Factura(5, 4);
otraFactura.muestraId();
```


JavaScript Avanzado (XVII)

- Incluir los métodos de los objetos como funciones dentro de la propia función constructora, es una técnica que funciona correctamente pero que tiene un gran inconveniente que la hace poco aconsejable
- En el ejemplo anterior, las funciones **muestraCliente()** y **muestraId()** se crean de nuevo por cada objeto creado
- Con esta técnica, cada vez que se instancia un objeto, se definen tantas nuevas funciones como métodos incluya la función constructora

JavaScript Avanzado (XVIII)

- La penalización en el rendimiento y el consumo excesivo de recursos de esta técnica puede suponer un inconveniente en las aplicaciones profesionales realizadas con **JavaScript**
- **JavaScript** incluye una propiedad que no está presente en otros lenguajes de programación y que soluciona este inconveniente. La propiedad se conoce con el nombre de **prototype** y es una de las características más poderosas de JavaScript

JavaScript Avanzado (XIX)

- Cada tipo de objeto diferente hereda de un objeto ***prototype*** diferente
- En cierto modo, se puede decir que el **prototype** es el molde con el que se fabrica cada objeto de ese tipo. Si se modifica el molde o se le añaden nuevas características, todos los objetos fabricados con ese molde tendrán esas características
- Normalmente los métodos no varían de un objeto a otro del mismo tipo, por lo que se puede evitar el problema de rendimiento comentado anteriormente añadiendo los métodos al prototipo a partir del cual se crean los objetos

JavaScript Avanzado (XXI)

- La clase anterior que incluye los métodos en la función constructora, se puede reescribir utilizando el objeto **prototype**

```
function Factura(idFactura, idCliente) {
    this.idFactura = idFactura;
    this.idCliente = idCliente;
}
Factura.prototype.muestraCliente = function() {
    alert(this.idCliente);
}
Factura.prototype.muestraId = function() {
    alert(this.idFactura);
}
```

Se han añadido los dos métodos del objeto en su prototipo

De esta forma, todos los objetos creados con esta función constructora incluyen por defecto estos dos métodos

Además, no se crean dos nuevas funciones por cada objeto, sino que se definen únicamente dos funciones para todos los objetos creados

JavaScript Avanzado (XXII)

- En el **prototype** de un objeto sólo se deben añadir aquellos elementos comunes para todos los objetos
- Normalmente se añaden los métodos y las constantes. Las propiedades del objeto permanecen en la **función constructora** para que cada objeto diferente pueda tener un valor distinto en esas propiedades
- El mayor inconveniente de la propiedad **prototype** es que se comparte entre todos los objetos y puede reescribir propiedades y métodos de forma accidental

```
Factura.prototype.iva = 16;
var laFactura = new Factura(3, 7); // laFactura.iva = 16
Factura.prototype.iva = 7;
var otraFactura = new Factura(5, 4);
// Ahora, laFactura.iva = otraFactura.iva = 7
```

JavaScript Avanzado (XXV)

- **Excepciones: JavaScript** dispone de un mecanismo de tratamiento de excepciones muy similar al de otros lenguajes de programación. Para ello, define las palabras reservadas **try**, **catch** y **finally**
- La palabra reservada **try** se utiliza para encerrar el bloque de código **JavaScript** en el que se van a controlar las excepciones. El bloque definido por **try** va seguido de otro bloque de código definido por **catch**
- Cuando se produce una excepción en el bloque **try**, se ejecutan las instrucciones contenidas en el bloque **catch**. Después del bloque catch, es posible definir un bloque con la palabra reservada **finally**. Todo el código contenido en el bloque **finally** se ejecuta independientemente de la excepción ocurrida en el bloque **try**

JavaScript Avanzado (XXVI)

- A continuación se muestra un ejemplo de excepción y uso de los bloques **try** y **catch**

```
try {
    var resultado = 5/a;
} catch(excepcion) {
    alert(excepcion);
}
```

JavaScript también permite lanzar excepciones manualmente mediante la palabra reservada **throw**

```
try {
    if(typeof a == "undefined" || isNaN(a)) {
        throw new Error('La variable "a" no es un número');
    }
    var resultado = 5/a;
} catch(excepcion) {
    alert(excepcion);
} finally {
    alert("Se ejecuta");
}
```

JavaScript Avanzado (XXVIII)

- **Reflexión:** La reflexión es un proceso mediante el cual un programa es capaz de obtener información sobre sí mismo y por tanto es capaz de auto ***modificarse*** en tiempo de ejecución
- **JavaScript** define mecanismos que permiten la reflexión sobre los objetos para permitir descubrir propiedades y métodos de objetos externos

```
if(elObjeto.laPropiedad) {
    // el objeto posee la propiedad buscada
}

if(typeof(elObjeto.laPropiedad) != 'undefined') {
    // el objeto posee la propiedad buscada
}

if(elObjeto instanceof Factura) {
    alert("Se trata de un objeto de tipo Factura");
}
```


JavaScript Avanzado (XXVIII)

Paso por valor vs paso por referencia

- En JS los atributos primitivos (string, numero, boolean) siempre se pasan a las funciones por valor (el valor se copia dentro del método, pero su modificación no afecta al atributo externo).

```
function funcion(x)
{
    x = 2;
}

var x = 1;
funcion(x);

alert(x);
```

- Los objetos (incluidos los arrays) se pasan por referencia

```
var obj = new objeto();
obj.valor = 1;

function funcion (param1){
param1.valor = 2;
}
funcion (obj);

alert(obj.valor);  //¿Qué muestra?
```