Desarrollo Web en entorno Cliente: JAVASCRIPT

UNIDAD 9. CLASES/OBJETOS

UNIDAD 9. CLASES/OBJETOS

CLASES/OBJETOS EN JAVASCRIPT

En los capítulos anteriores se ha ofrecido una visión general sobre el **modelo de objetos de Javascript**. Se ha visto que **este lenguaje está basado en objetos** y hemos estudiado los objetos intrínsecos del lenguaje y los del navegador.

Aunque no está considerado como un lenguaje orientado a objetos al estilo de C++ o de Java, sin embargo, **es posible implementar los denominados objetos personalizados**, que hemos llamado, quizás exageradamente, clases.

Vamos a ver como se trabaja con clases/objetos. En Javascript, como en cualquier lenguaje, un objeto no es más que un conjunto de atributos y funciones (sus métodos). A cada objeto podemos añadirle nuevas propiedades y funciones de diferentes maneras que iremos viendo. Sin embargo **la herencia**, tal y como la conocemos de otros lenguajes como Java, no existe, pues no existen las clases: para crear un nuevo objeto que tenga los mismos atributos que otro, estos deben ser copiados. A esta forma de trabajar se llama **herencia basada en prototipos**. Esto significa que no hay clases o interfaces como en Java o C++: solo hay objetos que copian (heredan) sus propiedades de otros objetos, llamados **prototipos**. Así que olvidémonos de la clásica definición de "objeto como instancia de una clase", porque aquí no hay clases (y por esta razón, cuando se hable de "clases" en Javascript, lo haremos siempre teniendo en cuenta lo dicho), por lo tanto, nos referiremos a clases/objetos.

9.1 CONSTRUCTOR

Crear nuestros propios moldes o constructores para objetos será tan sencillo como hacer:

```
function NombreClase(dato_1, ..., dato_N)
  {
  this.propiedad_1 = dato_1;
  ....
  this.propiedad_N = dato_N;
  this.metodo_1 = Funcion_1;
   ....
  this.metodo_N = Funcion_N;
  }
```

Según lo dicho unos párrafos más arriba, ***NombreClase*** **es el nombre del constructor de la clase/objeto**. **this es un objeto especial** (**su valor hace referencia al propietario de la función que la está invocando o en su defecto, al objeto donde dicha función es un método**) direcciona el objeto actual que está siendo definido en la declaración. Es decir, se trata de una referencia al objeto actual (en este caso, el que se define). **Al definir los métodos, sólo ponemos el nombre, no sus argumentos**. La implementación de cada uno de los métodos de la clase/objeto se hará de la misma forma que se declaran las funciones, es decir, haremos:

```
function Funcion_1([arg_1], ..., [arg_M])
  {
  //cosas varias que haga la función
  }
```

y cuando vayamos a crear los objetos, tenemos que usar un operador especial **new**, seguido del constructor del objeto al que le pasamos como argumentos, los argumentos con los que hemos definido el molde. Así, haciendo:

```
var Objeto = new NombreClase(dato1, ..., datoN);
```

creamos el objeto *Objeto* según el molde NombreClase, y ya podemos usar sus métodos y modificar o usar convenientemente sus atributos o propiedades.

Las funciones/métodos definidas para cualquier clase/objeto se deben llamar de diferente manera. En la misma clase/objeto no puede haber dos métodos que se llamen igual. **No está definida la sobrecarga en Javascript**.

> **Definición de Sobrecarga** es la capacidad de un lenguaje de programación, que permite nombrar con el mismo identificador diferentes variables u operaciones.
>
> En programación orientada a objetos la sobrecarga se refiere a la posibilidad de tener dos o más funciones con el mismo nombre pero funcionalidad diferente. Es decir, dos o más funciones con el mismo nombre realizan acciones diferentes. El compilador usará una u otra dependiendo de los parámetros usados. A esto se llama también sobrecarga de funciones.
>
> También existe la sobrecarga de operadores que al igual que con la sobrecarga de funciones se le da más de una implementación a un operador.

Por ejemplo el tratamiento de la clase/objeto Persona podría ser de la siguiente manera:

```
<html>
<head>
  <script language="javascript">
    <!--
    function Persona(pNombre,pEdad,pTelefono)
      {
      this.Nombre=pNombre;
      this.Edad=pEdad;
      this.Telefono=pTelefono;
      this.toString=Persona_toString;
      this.Prefijo=Persona_Prefijo;
      }

    function Persona_toString()
      {
      return (this.Nombre+" "+this.Edad+" "+this.Telefono);
      }

    function Persona_Prefijo(n)
      {
      return (this.Telefono.substr(0,n));
      }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
```

```
    oPersona1=new Persona('Pepe',45,'928111111');
    document.write("<h2>Datos de "+oPersona1.Nombre+"</h2>");
    document.write("Edad: "+oPersona1.Edad+"<br>");
    document.write("Tlf: "+oPersona1.Telefono+"<br>");
    document.write(oPersona1.toString()+"<br>");
    document.write("Prefijo: "+oPersona1.Prefijo(3));

    oPersona2=new Persona('Pepa',25,'912222222');
    document.write("<h2>Datos de "+oPersona2.Nombre+"</h2>");
    document.write("Edad: "+oPersona2.Edad+"<br>");
    document.write("Tlf: "+oPersona2.Telefono+"<br>");
    document.write(oPersona2+"<br>");
    document.write("Prefijo: "+oPersona2.Prefijo(2));
    -->
  </script>
</body>
</html>
```

Persona.html **Persona -Int_datos.html** (sólo profe no en ejemplos)

Ejercicio alumnado: Probar las características de la clase persona, o sea, ejemplo anterior.

En Javascript podemos acceder a las propiedades de un objeto usando la notación ***objeto.propiedad;*** o de manera indexada como un array o con ***objeto[prop]***

Acceso normal

```
document.write("hola "+obj1.nombre);
```

Acceso indexado con un literal

```
document.write("hola "+obj1[ "nombre" ]);
```

Acceso indexado con una variable

```
var prop = "nombre";
document.write("hola "+obj1[ prop ])
```

ACCESO A PROPIEDADES

Invocación normal del método

```
obj1.saluda();
```

Acceso al método indexado con un literal

```
obj1[ "saluda" ]();
```

Acceso al método indexado con una variable

```
var func = "saluda";
obj1[ func ]();
```

ACCESO A MÉTODOS

Persona-acceso_indexado_como_un_array.html (sólo profe no en ejemplos)

Un atributo de objeto puede ser a su vez otro objeto:

```
<html>
<head>
  <script language="javascript">
    <!--
    function Persona(pNombre,pEdad,pTelefono)
      {
      this.Nombre=pNombre;
      this.Edad=pEdad;
      this.Telefono=pTelefono;
      this.toString=Persona_toString;
      this.Prefijo=Persona_Prefijo;
      }

    function Persona_toString()
      {
      return (this.Nombre+" "+this.Edad+" "+this.Telefono);
      }

    function Persona_Prefijo(n)
      {
      return (this.Telefono.substr(0,n));
      }

    function Coche(pMatricula,pMarca,pModelo,pPropietario)
      {
      this.Matricula=pMatricula;
      this.Marca=pMarca;
      this.Modelo=pModelo;
      this.Propietario=pPropietario;
      this.toString=Coche_toString;
      }

    function Coche_toString()
     {
     return (this.Matricula+" "+this.Marca+" "+this.Modelo+"<br>"+
               "<b>propietario:</b> "+this.Propietario);
     }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
    oPersona=new Persona('Pepe',45,'928111111');
    document.write("<h2>Datos de la persona "+oPersona.Nombre+"</h2>");
    document.write("Edad: "+oPersona.Edad+"<br>");
    document.write("Tlf: "+oPersona.Telefono+"<br>");
    document.write(oPersona+"<br>");
    document.write("Prefijo: "+oPersona.Prefijo(3));

    oCoche=new Coche('1234RTS','Audi','TT',oPersona);
    document.write("<h2>Datos del coche "+oCoche.Matricula+"</h2>");
    document.write(oCoche+"<br>");
    -->
  </script>
</body>
</html>
```

Coche.html (sólo profe no en ejemplos)

Ejercicio alumnado: Probar el ejemplo anterior, características de clase coche.

Método toString: Devuelve una cadena que representa al objeto.

El método toString() está ya incluido en la librería de programación y nos sirve para mostrar el contenido de un objeto deseado. Este puede ser modificado y nos puede ser de gran ayuda.

Todos los objetos tienen un método toString que se llama automáticamente cuando el objeto se representa como un valor de texto o cuando un objeto se referencia de tal manera que se espera una cadena. Por defecto, el método toString es heredado por todos los objetos que descienden de Object.

Si este método no se sobreescribe en el objeto personalizado, toString devuelve [object type], donde type es el tipo de objeto.
El siguiente código ilustra esto:

```
var objeto = new Object();
objeto.toString();          // Devuelve [object Object]
```

Sobreescribir el método por defecto toString
Podemos crear una función que sea llamada en lugar del método predeterminado toString. El método toString no toma argumentos y debería devolver una cadena. El método toString que creemos puede ser cualquier valor que queramos, pero será más útil si aportamos información sobre el objeto.

Otro ejemplo:

```
function Saludator(nom) {
        alert("Saludos factory!!")
        this.nombre = nom+"s"
        this.saluda = function() {
                        alert("hola "+this.nombre)
                      }
}

var obj1 = new Saludator("mundo")
var obj2 = new Saludator("gente")
obj1.saluda() // "hola mundos"
obj2.saluda() // "hola gentes"
```

La función Saludator() es el constructor del objeto, y podemos pasarle parámetros que posteriormente podemos usar en los atributos del objeto.

Saludator.htm (sólo profe no en ejemplos)

Dentro de la función saluda sigue siendo necesario usar this.nombre para acceder al atributo nombre.

Podemos crear nuestros objetos usando funciones como expresiones, y no como declaraciones. El siguiente código es completamente equivalente al anterior:

```
var  Saludator = function(nom) {
                  alert("Saludos factory!!")
                  this.nombre = nom+"s"
                  this.saluda = function() {
                                    alert("hola "+this.nombre)

                                }
                  }
```

Es necesario señalar que las funciones que servirán para crear objetos se suelen nombrar con la primera letra en mayúscula. Esto es así para evitar que se confundan con funciones normales o variables.

9.2 OTRAS MANERAS DE CREAR OBJETOS SIN USAR function nombre_clase

Existen otras maneras de crear objetos sin usar function. Estudiemos, al menos, dos de ellas:

```
var obj1 = new Object()

var obj2 = {}
```

Para añadir una nueva propiedad, sólo tenemos que asignársela:

```
var obj1 = new Object()
obj1.nombre = "mundo"

alert("hola "+obj1.nombre); // hola mundo
```

Podemos también eliminar una propiedad usando el **operador delete**:

```
var  obj1 = new Object()
alert(obj1.nombre)   // undefined

obj1.nombre = "mundo"
alert(obj1.nombre)   // mundo

delete obj1.nombre
alert(obj1.nombre)   // undefined otra vez
```

Y para añadir un método, solo hay que añadirle una función como expresión, teniendo siempre en cuenta que para que un método pueda acceder a las propiedades del objeto donde reside, tenemos que usar el prefijo this:

```
var obj1 = new Object()
obj1.nombre = "mundo"
obj1.saluda = function() { document.write("hola "+this.nombre) }
obj1.saluda();
```

Aquí vemos como es necesario usar this.nombre dentro de la función saluda() para acceder al atributo nombre del objeto.

Otra forma de construir objetos es como literales:

```
var obj1 = {nombre:"mundo",
            saluda: function() { document.write("hola "+this.nombre)}
           };
obj1.saluda();
```

Como vimos anteriormente en Javascript podemos acceder a las propiedades de un objeto usando la notación **objeto.propiedad**; o de manera indexada como un array, con **objeto[prop]** . En este segundo caso, **prop** es una variable de tipo String cuyo contenido es el nombre de la propiedad a acceder.

Ejemplo:

```
var obj1 = new Object()
obj1.nombre = "mundo"
obj1.saluda = function() { alert("hola "+this.nombre) }


alert("hola "+obj1.nombre)       // Acceso normal

alert("hola "+obj1[ "nombre"])  // Acceso indexado con un literal

var prop = "nombre"
alert("hola "+obj1[ prop ])      // Acceso indexado con una variable


obj1.saluda()          // Invocación normal

obj1[ "saluda"]()    // Acceso indexado con un literal

var func = "saluda"
obj1[ func ]()         // Acceso indexado con una variable
```

Podemos usar esta segunda forma de **acceder a las propiedades de un objeto para listarlas todas, sin importar cuantas tenga ni su nombre**:

```
var obj1 = new Object()
obj1.nombre = "mundo"
obj1.fecha = new Date()
obj1.luckyNumbers = [4,8,15,16,23,42]
obj1.saluda = function() { alert("holas "+this.nombre) }

var name
```

```
for (name in obj1) {
      alert(name+" : "+ obj1[name])
}
```

Acceder_propiedades_de_un_objeto.html (sólo profe no en ejemplos)

Ejercicio alumnado: Probar los ejemplos anteriores.

9.3 PROPIEDAD prototype

A través de la **propiedad prototype que tienen todas las clases/objetos podemos agregar nuevas propiedades y métodos:**

```
var Saludator = function(nom) {
                        this.nombre = nom
                        this.saluda = function() {
                                        document.write("hola "+this.nombre)
                                        }
                }
```

Añadimos nuevos atributos y métodos a través de prototype

```
Saludator.prototype = {
        apellido: "cruel",
        despide: function() {
                        document.write("adios "+this.nombre+" "+this.apellido)
                  }
}
var obj1 = new Saludator("mundo")
obj1.despide()                  // "adios mundo cruel"
obj1.apellido="en crisis";
obj1.despide()                  // "adios mundo en crisis"
```

El siguiente ejemplo es completamente equivalente al anterior:

```
var Saludator = function(nom) {
                    this.nombre = nom
                 }
```

Añade nuevos atributos y métodos a través de prototype

```
Saludator.prototype.apellido = "cruel"
Saludator.prototype.despide = function() {
                document.write("adios "+this.nombre+" "+this.apellido)
                }
var obj1 = new Saludator("mundo")
obj1.despide() // "adios mundo cruel"
obj1.apellido="en crisis";
obj1.despide() // "adios mundo en crisis"
```

Finalmente decir que **podemos acceder al atributo prototype en todas las clases**, incluso en las **clases del sistema**, como String o Date. El siguiente código añade una nueva versión toString() a la clase de sistema Date:

```
<script language="javascript">
  <!--
  Ahora=new Date();
  document.write("Salida por defecto: "+Ahora+"<br>");

  function Date_toString()
    {
    var sFecha;
    sFecha=this.getDate()+"/"+(this.getMonth()+1)+"/"+this.getFullYear();
    return sFecha;
    }
  Date.prototype.toString=Date_toString;
  document.write("Salida personalizada: "+Ahora);
  -->
</script>
```

Ejemplo anterior (Añadir características a una clase existente**)**: **Prototype.html**

9.4 ARRAY DE OBJETOS

Los ejemplos de personas y coches pueden implementarse haciendo que los objetos de tipo coche sean elementos de un array, de forma que para imprimir sus propiedades se use un bucle for. En la página **ArrayCoches.htm** se ha introducido esta modificación:

```
<html>
<head>
  <script language="javascript">
    <!--
    function Persona(pNombre,pApellidos,pDireccion)
      {
      this.Nombre=pNombre;
      this.Apellidos=pApellidos;
      this.Direccion=pDireccion;
      this.toString=Persona_toString;
      this.NombreCompleto=Persona_NombreCompleto;
      }

    function Persona_toString()
      {
      return (this.Nombre+" "+this.Apellidos+" "+this.Direccion);
      }

    function Persona_NombreCompleto()
      {
      return (this.Apellidos+" "+this.Nombre);
      }
```

```
    function Coche(pMatricula,pMarca,pModelo,pPropietario)
      {
      this.Matricula=pMatricula;
      this.Marca=pMarca;
      this.Modelo=pModelo;
      this.Propietario=pPropietario;
      this.toString=Coche_toString;
      }


     function Coche_toString()
      {
       return (this.Matricula+" "+this.Marca+" "+this.Modelo+"<br>"+
            "<b>propietario: "+this.Propietario);
      }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
    var coches=new Array();
    var Jorge=new Persona("Jorge","Washington Usa","Casa Blanca s/n");
    var Barbara=new Persona("Bárbara","Streissand Cine",
                                    "c/ Hollywood 345, 6° B");
    coches[0]=new Coche("GC1234SG","Audi","TT",Jorge);
    coches[1]=new Coche("1234SDF","Volkwagen","Golf GTI",Bárbara);
    document.write("<h2>Listado de coches</h2>");
    document.write("<table border='1'>");
    for (var i=0;i<coches.length;i++)
      {
      document.write("<tr><td>",coches[i],"</td>");
      document.write("</tr>");
      }
    document.write("</table>");
    document.write("<h2>Otro listado de coches</h2>");
    document.write("<table border='1'>");
    for (var i=0;i<coches.length;i++)
      {
      document.write("<tr><td>",
                     coches[i].Matricula,"</td>");
      document.write("<td>",coches[i].Marca,"</td>");
      document.write("<td>",coches[i].Modelo,"</td>");
      document.write("<td>",coches[i].Propietario,"</td>");
      document.write("</tr>");
      }
    document.write("</table>");
    document.write("<h2>Otro listado de coches más</h2>");
    document.write("<table border='1'>");
    for (var i=0;i<coches.length;i++)
      {
      document.write("<tr><td>",
                     coches[i].Matricula,"</td>");
      document.write("<td>",coches[i].Marca,"</td>");
      document.write("<td>",coches[i].Modelo,"</td>");
      document.write("<td>",coches[i].Propietario.NombreCompleto(),
                    "</td>");
      document.write("</tr>");
      }
```

```
    document.write("</table>");
    document.write("<h2>No se cuantos listado de coches</h2>");
    document.write("<table border='1'>");
    for (var i=0;i<coches.length;i++)
      {
      document.write("<tr><td>",
                     coches[i].Matricula,"</td>");
      document.write("<td>",coches[i].Marca,"</td>");
      document.write("<td>",coches[i].Modelo,"</td>");
      document.write("<td>",coches[i].Propietario.Nombre,
                     "</td>");
      document.write("<td>",coches[i].Propietario.Apellidos,
                     "</td>");
      document.write("<td>",coches[i].Propietario.Direccion,
                     "</td>");
      document.write("</tr>");
      }
    document.write("</table>");
    -->
  </script>
</body>
</html>
```

Ejemplo anterior (Array de objetos de la clase coche**)**: **ArrayCoches.html**

9.5 MÉTODOS PRIVADOS

Podemos crear métodos privados (es decir, métodos que son sólo accesibles desde dentro del objeto, nunca desde fuera), creando funciones como declaración, no como expresión. La principal desventaja es que estos tipos de métodos no tienen ligado el ***this*** con el objeto actual, por lo que no pueden acceder a sus datos internos. Para evitar esto, tenemos que enviar como parámetros los datos del objeto que necesiten. Además, no podemos usar el prefijo ***this*** para llamar a estos métodos:

```
var Saludator = function(nom) {
        this.nombre = nom;
        this.saluda = function() {
                document.write("hola "+plural(this.nombre));
        }
        function plural(n) {
                return n + "s";
        }
}
var obj1 = new Saludator("mundo")
obj1.saluda()
// obj1.plural()  // fallaría porque plural() es privado
```

La llamada al método plural() no es accesible desde fuera, por lo que fallaría (por eso está comentada). Observemos también como la llamada a ***plural()*** no se hace con ***this.plural()***.

Y que para acceder al atributo nombre hemos tenido que pasárselo como parámetro.

Ciertamente resulta muy artificioso.

Los métodos privados no son dinámicos, en el sentido que no se pueden definir condicionalmente en tiempo de ejecución. Así que el siguiente código no fallará, pero no funcionará correctamente (de hecho, su funcionamiento dependerá del navegador):

```
// Atención, codigo erróneo a modo de ejemplo, NO USAR!
var Saludator = function(nom, enPlural) {
                  this.nombre = nom
                  this.saluda = function() {
                                    alert("hola "+quien(this.nombre))
                                }
                  // NO HACER ESTO NUNCA!!!!!!
                  if (enPlural) {

                        function quien(n) {     // Arggg!!
                              return n + "s"
                        }
                  } else {
                        function quien(n) {     // Ouch!!
                              return n
                        }
                  }
            }
var obj1 = new Saludator("mundo", false)
var obj2 = new Saludator("mundo", true)
obj1.saluda() // "hola mundo"
obj2.saluda() // "hola mundo", pero es incorrecto!
```

En su lugar, definiendo los métodos con funciones como expresiones, si que funciona correctamente, ya que la creación del método no es más que una asignación que se produce dentro de una bifurcación condicional:

```
var Saludator = function(nom, enPlural) {
                        this.nombre = nom
                        this.saluda = function() {
                                          alert("hola "+this.quien())
                                      }
                        if (enPlural) {
                              this.quien = function() {
                                                return this.nombre + "s"   // En plural
                                           }
                        } else {
                              this.quien = function() {
                                                return this.nombre       // En singular!!
                                           }
                        }
                  }
var obj1 = new Saludator("mundo", false)
var obj2 = new Saludator("mundo", true)
obj1.saluda()           // "hola mundo"
obj2.saluda()           // "hola mundos", correcto!
```

9.6 MÉTODOS Y ATRIBUTOS DE CLASE O ESTÁTICOS

Son métodos o atributos comunes a todas las instancias de una clase (también llamados estáticos). Para crear un método estático tan solo tenemos que asignar a nuestra clase una función más:

```
var Saludator = function(nom, enPlural) {
        this.nombre = nom;
        this.enPlural = enPlural;
        this.saluda = function() {
                document.write("hola "+this.nombre)
        }
}
// Añadimos un método estático como atributo de la "clase" Saludator
Saludator.cloneSaludo = function(saludo) {
        return new Saludator("clonado "+saludo.nombre, saludo.enPlural)
}

var obj1 = new Saludator("mundo", false)
var obj2 = Saludator.cloneSaludo(obj1)
obj1.saluda() // "hola mundo"
obj2.saluda() // "hola clonado mundo"
```

Ejemplo anterior : met_atrib_clase_estaticos.html

Por supuesto, podemos añadir también atributos estáticos.

Todas las propiedades estáticas (atributos o métodos) son siempre públicas y necesitan SIEMPRE el prefijo con el nombre de la “clase” delante siempre para acceder a ellas.

En este ejemplo, se usa la propiedad estática ***count*** de la clase Saludator para llevar la cuenta del número de objetos que se han creado:

```
var Saludator = function(nom) {
        // Se inicializa count a 0, si no existe
        Saludator.count = Saludator.count?Saludator.count:0
        Saludator.count ++
        this.nombre = nom
        this.saluda = function() {
                document.write("hola "+this.nombre)
        }
}
Saludator.cloneSaludo = function(saludo) {
        return new Saludator("clonado "+saludo.nombre)
}

var obj1 = new Saludator("mundo")
var obj2 = Saludator.cloneSaludo(obj1)
obj2.saluda()
document.write(Saludator.count) // "2", pues se han creado 2 objetos
```

Metodos_atributos_estaticos.html (sólo profe no en ejemplos)

Propiedad_atributos_estaticos.html (sólo profe no en ejemplos)

9.7 HERENCIA: Métodos de function

Hacer POO con Javascript tiene más de artista de circo que de verdadera programación. Pero es lo que hay. Lo cierto es que, con paciencia y algunas ideas geniales lo podemos hacer casi todo. Las dos cosas más difíciles son la herencia y las variables de clase.

Las funciones son instancias de la clase/objeto function **y como tal tienen también métodos propios**. Los más utilizados son call() y apply().

La herencia se puede implementar con el ***método call()*** y el ***método apply()***. La única diferencia entre los dos métodos es la forma en la que se pasan los argumentos a la función.

El método **.call()** recibe los mismos argumentos que la función más uno, el valor que tendrá **this** que se pasa antes que los demás argumentos:

call([thisObj[, arg1[, arg2[, [, argN]]]]])

Parámetros: thisObj →Opcional. Objeto que se va a utilizar como objeto actual.
arg1, arg2, , argN →Opcional. Lista de argumentos que se van a pasar al método.

El método **.apply()** actúa de forma bastante similar a **.call()**, pero con una variación, solo recibe dos argumentos, el primero es el contexto de la función, el valor de **this** y el segundo será un array que contendrá los argumentos que se le pasarán a la función:

apply([thisObj[,argArray]])

Parámetros: thisObj → Opcional. Objeto que se va a usar como objeto this.
argArray → Opcional. Conjunto de argumentos que se van a pasar a la función.

Ambos permiten ejecutar una función como si fuera un método de otro objeto.

Veamos un ejemplo:

```
<html>
<head>
  <script language="javascript">
    <!--
    function Persona(pNombre,pApellidos,pDireccion)
      {
      this.Nombre=pNombre;
      this.Apellidos=pApellidos;
      this.Direccion=pDireccion;
      this.toString=Persona_toString;
      this.NombreCompleto=Persona_NombreCompleto;
      }

    function Persona_toString()
      {
      return (this.Nombre+" "+this.Apellidos+" "+this.Direccion);
      }

    function Persona_NombreCompleto()
```

```
        {
        return (this.Apellidos+" "+this.Nombre);
        }

      function Alumno(pNombre,pApellidos,pDireccion,pCurso)
        {
        Persona.call(this,pNombre,pApellidos,pDireccion);
        this.Curso=pCurso;
        this.toString=Alumno_toString;
        }

/*
      function Alumno(pNombre,pApellidos,pDireccion,pCurso)
        {
        Persona.apply(this,arguments);
        this.Curso=pCurso;
        this.toString=Alumno_toString;
        }
*/

      function Alumno_toString()
        {
        return (this.Nombre+" "+this.Apellidos+" "+this.Direccion+
                "<br>"+"<b>curso:</b> "+this.Curso);
        }
      -->
    </script>
  </head>
  <body>
    <script language="javascript">
      <!--
      var oPersona1=new Persona("Jorge","Washington Usa","Casa Blanca s/n");
      document.write(oPersona1);
      document.write("<hr>");
      var oAlumno1=new Alumno("Jorge","Negrete Blanco","Casa Morada n° 123",
                              "Bases de Datos");
      document.write(oAlumno1.toString());
      -->
    </script>
  </body>
  </html>
```

Ejemplo anterior (Implementar herencia en javascript**):**

Herencia_call.html **Herencia_apply.html**

Para más información ver los enlaces siguientes:

Método Apply(): http://msdn.microsoft.com/es-es/library/4zc42wh1(v=vs.94).aspx

Métodos Call() y Apply(): http://www.amatiasq.com/2012/01/javascript-conceptos-basicos-this-call-y-apply/

http://librosweb.es/ajax/capitulo_3.html

http://www.w3schools.com/js/js_function_invocation.asp

9.8 OTRAS FORMAS DE IMPLEMENTAR LA HERENCIA

Como es normal en Javascript, hay más de una forma de implementar herencia, esto es debido a que la herencia no es explícita, es emulada. Vamos a ver los dos métodos más utilizados.

Object masquerading

El problema de la herencia basada en prototipos es que la "***clase padre***" no es una clase, sino un objeto que se crea una única vez y que sirve de referencia para todas las instancias de la clase. Así que para poder heredar e invocar constructores en Javascript tenemos que hacerlo a mano.

Primero debemos crear una referencia al constructor de la clase padre y después llamarlo explícitamente:

```
var Saludator = function(nom) {
        this.nombre = nom
        this.saluda = function() {
                document.write("hola "+this.nombre)
        }
}
var SubSaludator = function(nom, ape) {
        // Primero se añade una referencia al constructor padre,
        // y después se llama explícitamente.
        this.superConstructor = Saludator
        this.superConstructor(nom)
        this.apellido = ape
        this.despide = function() {
                document.write("adios "+this.nombre+" "+this.apellido)
        }
}
var obj1 = new Saludator("mundo")
var obj2 = new SubSaludator("gente", "cruel")

obj1.saluda()  // "hola mundo"
obj2.saluda()  // "hola gente"
//obj1.despide() // error Saludator no tiene el método despide()
obj2.despide() // "adios gente cruel"
```

El inconveniente es que un objeto SubSaludator no se considera que sea también una instancia de Saludator.

```
document.write (obj2 instanceof(SubSaludator));       //true
document.write (obj2 instanceof(Saludator));          //false
```

Object Masquerading.html (sólo profe no en ejemplos)

Prototype chaining

Se realiza utilizando el atributo prototype.

Veamos un sencillo esquema:

```
function ClassA(){
          this.nombre="";
          this.identificarse = function(){
                                document.write(this.nombre);
                                        }
          }

function ClassB(){
          }

ClassB.prototype = new ClassA();
ClassB.prototype.saludar = function(){
                                document.write("Hola "+this.nombre);
                                }
  var a = new ClassA();
  a.nombre="clase A";
  a.identificarse();     //clase A

  var b=new ClassB();
  b.nombre="clase B";
  b.identificarse();     //clase B
  b.saludar();           //Hola clase B
```

Lo que ocurre es que asignamos al prototipo de '***ClassB***' una nueva instancia de '***ClassA***'.
A partir de ese momento, todos los objetos creados con '***ClassB***' van a tener, también, los mismos métodos y propiedades de la instancia de '***ClassA***'. Y si queremos agregar más métodos y propiedades, lo único que tenemos que hacer es agregárselos al prototype de '***ClassB***'.

Lo malo de este método para emular herencia, es que no se puede pasar parámetros a la clase base, como hicimos en el ejemplo de Object masquerading.

Lo bueno, es que el operador instanceof funciona de una manera única: por cada instancia de ***ClassB***, instanceof nos retorna true tanto con **ClassA** como con **ClassB**:

```
document.write (b instanceof(ClassB)); //true
document.write (b instanceof(ClassA)); //true
```

Prototype_chaining.html (sólo profe no en ejemplos)

Sin embargo podemos combinar ambos métodos y obtener todas las ventajas y ninguno de los inconvenientes:

```
var Saludator = function(nom) {
        this.nombre = nom
        this.saluda = function() {
                        document.write("hola "+this.nombre)
                        }
}
var SubSaludator = function(nom, ape) {
        // Primero se añade una referencia al constructor padre,
        // y después se llama explícitamente.
        this.superConstructor = Saludator
        this.superConstructor(nom)
        this.apellido = ape
        this.despide = function() {
                        document.write("adios "+this.nombre+" "+this.apellido)
                        }
}

SubSaludator.prototype = new Saludator()

var obj1 = new Saludator("mundo")
var obj2 = new SubSaludator("gente", "cruel")

obj1.saluda()  // "hola mundo"
obj2.saluda()  // "hola gente"
//obj1.despide() // error Saludator no tiene el método despide()
obj2.despide() // "adios gente cruel"

document.write (obj2 instanceof(Saludator)); //true
document.write (obj2 instanceof(SubSaludator)); //true
```

Implementar_herencia_completo.html (sólo profe no en ejemplos)