Parte 3
Aproximación a la POO en JavaScript

Capítulo 3-1
Enfoque orientado a "objetos" en JavaScript

1. Introducción

Aunque la implementación del modelo de programación orientada a objetos (POO) no esté tan completa en JavaScript como en C++ o Java, JavaScript ofrece los mecanismos principales gestionados por estos lenguajes.

Recordemos los conceptos más importantes de la POO:

– Encapsulación: reunión de un conjunto de propiedades (parte de tratamiento de datos) y de funciones, también llamadas métodos (parte de procesamientos), dentro de un objeto tipo (quizás es más correcto hablar de clase), con la posibilidad de crear (instanciar) objetos a partir de esta clase.

– Herencia: posibilidad de "fabricar" una nueva clase a partir de una clase existente; esta nueva clase hereda las propiedades y métodos de la clase padre (se pueden añadir nuevas propiedades/métodos a la nueva clase).

– Polimorfismo: un método del mismo nombre asociado a varias clases puede tener comportamientos diferentes para algunas de estas clases.

2. Programación orientada a objetos a través de ejemplos

JavaScript siempre ha sido una pieza esencial en los desarrollos web, principalmente para la programación del lado "cliente", es decir, del lado del navegador. Habitualmente, sin sumergirse de lleno en el lenguaje, los desarrolladores producen código JavaScript de calidad mediocre, contentándose con adaptar el código fuente recuperado de sitios web y manipulando los conceptos POO lo menos posible.

En paralelo, han aparecido un gran número de librerías JavaScript y su uso permite producir aplicaciones de mejor calidad. El dominio de estas librerías supone tener conocimientos básicos de POO en JavaScript.

Por tanto, el objetivo de la exposición que sigue es presentarle lo que hay que saber sobre este asunto. Los conceptos se van a explicar a través de una serie de ejemplos.

Algunos lectores que ya tengan una importante experiencia en otros lenguajes POO (PHP 5, Java, C++...) al principio se pueden sentir incómodos con los aspectos específicos de la POO en JavaScript, la POO por prototipado.

2.1 Secuencia 1: Declaración de los objetos JavaScript de manera "Inline"

Se trata de la manera más sencilla de declarar un objeto en JavaScript.

```
/* Declaración inline de un objeto JavaScript */
/* NB: Esta técnica no permite la herencia a partir del objeto
más adelante */
var Adicion = {
    x: 5,
    y: 10,
    calculo: function()
    {
        return this.x + this.y;
    }
};

/* Uso del objeto Adicion */
document.write("Suma: " + Adicion.calculo());
```

© Editions ENI - All rights reserved

El resultado obtenido de la ejecución será el siguiente:

```
Suma = 15
```

En este tipo de declaración de objeto, es posible prever la especificación de atributos (propiedades) y también de métodos.

La palabra clave `this` sirve para indicar que se está haciendo referencia a los atributos del objeto en sí mismo.

Está claro que con este tipo de declaración no será posible reutilizar este tipo de definición para crear un objeto de las mismas características (o parecidas). Por tanto, este método se utilizará poco (o nada) porque no permite la herencia. No se preocupe, porque volveremos sobre esto más en detalle, a lo largo de esta exposición.

2.2 Secuencia 2: Creación de objetos JavaScript con un constructor

También es posible crear nuestros objetos JavaScript con un constructor (concepto bien conocido en los lenguajes POO). En JavaScript, será suficiente con escribir una función y llamarla posteriormente con la palabra clave `new`. Por tanto, la función jugará el papel de clase sin serlo realmente.

Veamos con un ejemplo el desarrollo que es necesario seguir:

```
/* Definición de una función constructor, de nombre Coche */
var Coche = function()
{
    /* Atributo(s) del objeto */
    this.tieneMotor = true;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
        document.write("avanza");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Visualización del atributo tieneMotor del objeto simca1100 */
```

```
if (simca1100.tieneMotor)
{
    document.write("El coche simca1100 tiene un motor<br />");
}
else
{
    document.write("El coche simca1100 no tiene motor !<br />");
}

/* Llamada al método avanzar del objeto simca1100 */
document.write("El coche simca1100 ");
simca1100.avanzar();
```

En nuestro ejemplo, se define en primer lugar una función `Coche`. Integra un atributo booleano que indica que los coches tienen un motor y un método (función) de nombre `avanzar`, que mostrará "`avanza`" cuando se pida, a partir de un objeto de tipo `Coche` (se entenderá rápidamente).

Posteriormente, se construye un objeto de nombre `simca1100` a partir del constructor `Coche` (siento que el término "constructor" pueda ser confuso en un ejemplo basado en coches):

```
/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();
```

Ahora, la propiedad (atributo) `tieneMotor` se puede consultar para el objeto `simca1100` instanciado y también se puede ejecutar el método `avanzar`. Cuando se ejecute, tendremos:

```
El coche simca1100 tiene un motor
El coche simca1100 avanza
```

2.3 Secuencia 3: Variables privadas en una instancia de objeto

En el ejemplo de la secuencia anterior, habrá observado que las propiedades (atributos) llevan como prefijo la palabra clave `this`. Esto es lo que las hace usables desde el exterior (caso de la propiedad `tieneMotor`). Por el contrario, por necesidades locales del constructor (cálculo interno), puede declarar variables no expuestas, usando como prefijo la palabra clave `var`.

© Editions ENI - All rights reserved

Veamos un ejemplo concreto:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function()
{
    /* Variable(s) local(s) no accesible(s) desde el exterior del objeto */
    var numeroRuedas = 4;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
    document.write("avanza");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método avanzar del objeto simca1100 */
document.write("El coche simca1100 ");
simca1100.avanzar();

/* Intento de visualización de la variable local del constructor Coche */
document.write("<br />");
document.write("El coche simca1100 tiene " + simca1100.numeroRuedas +
" ruedas");
```

Se usa una variable local `numeroRuedas` en el constructor con el prefijo `var`. Esto solo es accesible, como estaba previsto, desde dentro del constructor, como se muestra en la ejecución de este script:

```
El coche simca1100 avanza
El coche simca1100 tiene undefined ruedas
```

2.4 Secuencia 4: Paso de argumento(s) a un constructor

En el ejemplo siguiente, vamos ver que es posible pasar uno o varios argumentos (lo habíamos visto ya para las funciones clásicas) a un constructor:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function(modelo)
{
    /* Atributo(s) del objeto informado durante la instanciación */
    this.modelo = modelo;
}
```

```
/* Instanciación de un objeto simca1100 a través del constructor Coche
con paso de argumento */
var miCoche = new Coche("simca1100");

/* Visualización del atributo modelo del objeto miCoche */
document.write("Tengo un " + miCoche.modelo);
```

El argumento `modelo` está en los paréntesis que siguen al nombre del constructor:

```
var Coche = function(modelo)
```

y está disponible en el cuerpo del constructor por:

```
    this.modelo = modelo;
```

A continuación, es suficiente a nivel de la instanciación del objeto `miCoche` con pasar como argumento un valor ("`simca1100`" en nuestro caso):

```
var miCoche = new Coche("simca1100");
```

La propiedad (atributo) `modelo` del objeto se mostrará por:

```
document.write("Tengo un " + miCoche.modelo);
```

2.5 Secuencia 5: No compartición de los métodos por las instancias de objetos

Dado que los métodos se declaran durante la instanciación de los objetos, sus definiciones están duplicadas en memoria.

En un ejemplo pequeño, el impacto es bajo.

Por el contrario, si su aplicación manipula muchos objetos con métodos múltiples y complejos en los constructores, esto se convierte en inmanejable.

El siguiente ejemplo destaca el problema que acabamos de comentar:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function()
{
    /* Atributo(s) del objeto */
    this.tieneMotor = true;
    /* Método(s) del objeto */
    this.avanzar = function()
```

© Editions ENI - All rights reserved

```
    {
        document.write("avanza");
    }
    this.retroceder = function()
    {
        document.write("retrocede");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Comprobación de la igualdad de métodos avanzar de los objetos simca1100
y renault12 */
if (simca1100.avanzar == renault.avanzar)
{
   document.write("Método avanzar compartido por los objetos simca1100 y
Renault12<br />");
}
else
{
    document.write("Método avanzar no compartido por los objetos simca1100
y Renault12<br />");
}
```

La ejecución confirma que el método `avanzar` no está factorizado:

```
Método avanzar no compartido por los objetos simca1100 y Renault12
```

La noción de prototipo que vamos a descubrir a continuación va a resolver este problema.

2.6 Secuencia 6: Noción de prototipo

Un prototipo es un conjunto de elementos (atributos/propiedades y métodos) que se va a asociar a un constructor (sin "almacenamiento" en el constructor en sí mismo). Durante la ejecución, cuando una propiedad de objeto solicitada en el código no se encuentre en el constructor del objeto en cuestión, se realizará una búsqueda en esta lista "adicional".

Capítulo 3
Adoptar las buenas prácticas

1. Espacio de nombres

1.1 Aspectos básicos

Al desarrollar, siempre evitaremos exponer demasiadas funciones o variables en el espacio de nombres global, para evitar conflictos de nombres. Esto significa dar acceso solo a lo que es útil y ocultar todo lo demás. Es parecido al principio de una caja negra con entradas y salidas bien definidas, pero la mecánica interna permanece oculta.

Esto es aún más importante si es probable que el código se utilice en otros contextos. El problema es que estamos acostumbrados a agregar funciones a nuestros archivos, sin considerar la reutilización. A medida que nuestro proyecto se vuelve más sustancial, se hace cada vez más difícil de gestionar. La asociación de dos códigos puede causar conflictos que no son necesariamente visibles inicialmente y el resultado se vuelve incierto.

Para limitar las colisiones de nombres, agregaremos un contexto adicional (una especie de super contexto), que asegurará que nuestras funciones y variables no se puedan alterar/usar accidentalmente. Este espacio de nombres es una especie de contenedor de nombres. De esta manera, un nombre solo tiene significado en relación con su espacio de nombres. Por lo tanto, la invocación de una función que no está en el espacio de nombres esperado, no es posible.

Como cada código tiene su propio espacio de nombres, ya no existe ningún riesgo de colisiones asociado con el código que pertenece a otro espacio de nombres.

Un espacio de nombres es una posibilidad asociada con muchos lenguajes. En el lenguaje Java, por ejemplo, la palabra clave `package` sirve para para realizar la declaración, de manera similar a como se hace en C# con `namespace`. En JavaScript, lamentablemente no tenemos una palabra clave para este uso, pero tenemos otros trucos igualmente poderosos para conseguirlo.

Este principio es importante para la calidad de sus programas. Una vez que lo entienda, se volverá natural y su desarrollo mejorará aún más.

Cuanto más grande sea su programa, más espacios de nombres necesitará. Es la dimensión que impone su uso. Por tanto, tener en cuenta el espacio de nombres será la garantía de un código de mejor calidad, que podrá evolucionar de forma más sencilla y con más seguridad.

1.2 Función

1.2.1 Función interna

Todas las declaraciones realizadas en una función, se vuelven locales para esa función. De hecho, como recordatorio, cuando se declara una variable en una función a través de la palabra clave `var`, su contexto de ejecución está vinculado al de la función y, fuera de la función, ya no existe. Por lo tanto, podemos prever que la función es un medio poderoso para limitar el alcance de las declaraciones y, por lo tanto, parece un candidato ideal para el espacio de nombres.

Ejemplo

```
function areaRectangulo( altura, anchura ) {
  var area = altura * anchura;
  alert( area + " cm2" );
}
areaRectangulo( 10, 5 );
alert( area );
```

© Editions ENI - All rights reserved

En este ejemplo, la variable `area` se declara dentro de la función `areaRectangulo`, por lo que no existe fuera de este perímetro. Por lo tanto, falla la última instrucción `alert(area)`. Nuestra función `areaRectangulo` actúa como un espacio de nombres.

Esta característica no se limita a las declaraciones de variables, sino que también es válida para las declaraciones de funciones.

Ejemplo

```
function area( tipo, altura, anchura ) {

  var area = 0;

  function areaRectangulo( altura, anchura ) {
    var area = ( altura * anchura );
    return area;
  }

  function areaCuadrado( lado ) {
    return areaRectangulo( lado, lado );
  }

  if ( tipo == "rectangulo" ) {
    area = areaRectangulo( altura, anchura );
  } else
  if ( tipo == "cuadrado" ) {
    area = areaCuadrado( altura );
  }

  return area;

}

alert( area( "rectangulo", 10, 20 ) );
alert( area( "cuadrado", 10 ) );
alert( areaCuadrado( 20 ) );        // ¡Error!
```

Las funciones `areaRectangulo` y `areaCuadrado` no existen fuera de la función `area`. Prueba de esto es que nuestra última instrucción para usar la función `areaCuadrado` provocó un error en tiempo de ejecución.

Al combinar estas funciones internas con variables locales, en realidad obtenemos un espacio de nombres para `areaRectangulo` y `areaCuadrado`, que solo pertenecen a la función `area`. Por lo tanto, nadie puede alterar o manipular estas funciones fuera de su función. Por tanto, es una técnica sencilla para eliminar los accesos no deseados.

1.2.2 Función anónima

Si queremos proteger nuestra ejecución, también es posible utilizar una función anónima como vimos brevemente en el primer capítulo. Esta función sin nombre garantizará un contexto independiente.

Ejemplo

```
(function () { // Nuestra función anónima

  function areaRectangulo( altura, anchura ) {
    var area = ( altura * anchura );
    return area;
  }

  function areaCuadrado( lado ) {
    return areaRectangulo( lado, lado );
  }

  alert( areaRectangulo( 10, 20 ) );
  alert( areaCuadrado( 20 ) );
} )();

alert( areaRectangulo( 10, 20 ) ); // Error
```

La función anónima se ejecuta tan pronto como se declara ya que, al no tener nombre, no se puede invocar posteriormente. Su función no es proporcionar un servicio reutilizable, sino proteger el contenido de cualquier acceso posterior.

La función anónima sirve como espacio de nombres para nuestro código. Todas las declaraciones internas ya no existirán después de que se ejecute, por lo que falla la invocación posterior del método `areaRectangulo`.

© Editions ENI - All rights reserved

La ejecución es saludable, porque no podemos crear una colisión en nuestros nombres de nuestras funciones. Si asociamos nuestro código con otro código que también definió la función `areaRectangulo`, nuestro código seguirá funcionando normalmente.

1.2.3 Función anónima con argumentos

El caso anterior es simple pero insuficiente en la práctica, porque nos gustaría tener una función `area` accesible desde cualquier parte, por ejemplo, pero sin hacer visibles las funciones anexas `areaCuadrado` y `areaRectangulo`.

Para ello, podemos crear un objeto vacío que servirá como contenedor para las funciones accesibles. A continuación, actuará como espacio de nombres para estas últimas.

Ejemplo

```
var miArea = {};

(function ( ns ) {
  function areaRectangulo( altura, anchura ) {
    var area = ( altura * anchura );
    return area;
  }
  function areaCuadrado( lado ) {
    return areaRectangulo( lado, lado );
  }
  function area() {
    if ( arguments.length == 2 ) {
      return areaRectangulo( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return areaCuadrado( arguments[ 0 ] );
    } else
    return "cálculo del área imposible";
  }

  ns.area = area; // Contenido accesible
} )( miArea );

alert( miArea.area( 10,20 ) );
```

En este ejemplo, hemos creado una función `área` interna en una función anónima. Esta utiliza las funciones `areaCuadrado` y `areaRectangulo`. Para que la función `area` se pueda usar en todas partes, lo hicimos para que la función anónima que se ejecuta pudiera tener un parámetro de objeto, que almacena la referencia al método `area`.

Este objeto es el que actuará como un espacio de nombres para el método `area`. Los otros métodos `areaCuadrado` y `areaRectangulo`, permanecen ocultos permanentemente para el usuario.

También podríamos haber hecho una prueba de la propiedad `area` del objeto al final de la función anónima, para no sobrescribir una declaración previa, por ejemplo con:

```
if ( !ns.area )
   ns.area = area;
```

Así, vemos que este sistema es similar a los métodos públicos y privado en la programación orientada a objetos. La función anónima entonces juega el papel de clase.

El hecho de que el método `area` pueda continuar funcionando fuera de la función anónima, también es un principio de cierre. El cierre es la capacidad de una función a la que se puede acceder fácilmente, para utilizar un contexto que ya no está disponible (por lo tanto, es privado).

La variable `miArea` está en el espacio de nombres global. Sin embargo, es posible reducir ligeramente los conflictos de nombres, utilizando el objeto predefinido `window`. Este objeto tiene la particularidad de tener sus propiedades visibles en el espacio de nombres global. Al reemplazar `miArea` por `window.miArea`, reducimos la visibilidad de nuestra variable `miArea`, que luego se convierte en una propiedad de `window`. Cuando se utiliza, no hay mucha diferencia.

De una manera general, la asociación de nuevas facultades a `window` depende más del contexto web. Intentaremos añadir facultades adicionales a nuestra ventana.

© Editions ENI - All rights reserved

Como observación, las colisiones de nombres están limitadas por los espacios de nombres, pero siempre con la condición de que estos también se distingan. De lo contrario, llevamos el problema a un nivel superior. Por lo general, se utilizará una convención para elegir un nombre de espacio de nombres de tipo URI (*Uniform Resource Identifier*), para asegurarse de que son los únicos titulares. Por ejemplo, si trabaja en una empresa ABC, se podría hacer que el objeto `miArea` se llamara `abc`, incluso `com_abc`.

1.3 Cierre

Ya hemos usado un cierre para nuestra función `area`, pero para ser más explícitos, aquí lo vamos a usar de otra manera, dejando al final solo una función de acceso.

```
var miArea = ( function () {
  function areaRectangulo( altura, anchura ) {
    var area = ( altura * anchura );
    return area;
  }
  function areaCuadrado( lado ) {
    return areaRectangulo( lado, lado );
  }
  function area() {
    if ( arguments.length == 2 ) {
      return areaRectangulo( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return areaCuadrado( arguments[ 0 ] );
    } else
    return "cálculo del área imposible";
  }
  return function() {
    return area.apply(this,arguments);
  };
} )();

alert( miArea( 20 ) );
alert( miArea( 10,20 ) );
```