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());
```

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.

© Editions ENI - All rights reserved

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`.

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
```

© Editions ENI - All rights reserved

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()
    {
        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.

© Editions ENI - All rights reserved

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".

Veamos un ejemplo completo:

```
/* Definición de una función constructor de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Comprobación de la existencia por defecto de un prototipo para
cualquier constructor */
document.write("Prototipo del constructor: " + Coche.prototype);

/* Añadir un método zigzaguear al prototipo del constructor Coche */
Coche.prototype.zigzaguear = function()
{
    document.write("zigzaguea peligrosamente<br />");
};

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

/* Llamada al método zigzaguear del objeto simca1100, accesible a través
del prototipo del constructor Coche */
document.write("<br />¿Qué hace el simca1100? ");
simca1100.zigzaguear();

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

/* Comprobación del método zigzaguear compartido o no, entre los objetos
simca1100 y renault12 */
if (simca1100.zigzaguear == renault12.zigzaguear)
{
    document.write("Método zigzaguear compartido por los objetos
simca1100 y renault12<br />");
}
else
{
    document.write("Método zigzaguear no compartido por los objetos
simca1100 y renault12<br />");
}
```

La opción que se ha utilizado en el ejemplo es codificar un constructor vacío. Después, se realiza una comprobación para demostrar que cualquier constructor tiene un prototipo:

```
document.write("Prototipo del constructor: " + Coche.prototype);
```

Posteriormente, se ha asociado un método `zigzaguear` al prototipo relacionado con el constructor `Coche`:

```
Coche.prototype.zigzaguear = function()
{
    document.write("zigzaguea peligrosamente<br />");
};
```

Después de la instanciación habitual de un objeto `simca1100` a partir del constructor `Coche`, se hace una llamada al método `zigzaguear` para el objeto `simca1100`.

También se instancia un segundo objeto, `renault12`, a partir del constructor `Coche` para demostrar que esta vez el método `zigzaguear` está compartido (es común a ambos objetos).

La ejecución da:

```
Prototipo del constructor: [object Object]
¿Qué hace el simca1100? Zigzaguea peligrosamente
Método zigzaguear compartido por los objetos simca1100 y renault12
```

2.7 Secuencia 7: Sobrecarga de un método

Para una instancia dada de objeto, es posible sobrecargar (modificar) un método (o una propiedad/atributo).

Veamos un ejemplo concreto de cómo aplicarlo:

```
/* Definición de una función constructor, de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Añadir un método cargar al prototipo del constructor Coche */
Coche.prototype.cargar = function()
{
    document.write("carga<br />");
};

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

/* Llamada al método cargar del objeto simca1100, accesible a través
del prototipo del constructor Coche */
document.write("El simca1100 ");
```

© Editions ENI - All rights reserved

```
simca1100.cargar();

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

/* Modificación (sobrecarga) del método cargar para el objeto
simca1100 */
simca1100.cargar = function()
{
    document.write("carga rápidamente <br />");
};

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

/* Llamada al método cargar (no sobrecargado) del objeto renault12 */
document.write("El renault12 ");
renault12.cargar();
```

La ejecución da como resultado:

```
El simca1100 carga
El simca1100 carga rápidamente
El renault12 carga
```

Puede comprobar que la modificación (sobrecarga) solo impacta al objeto `simca1100`.

2.8 Secuencia 8: Extensión de un prototipo

La extensión es la suma posterior de un método adicional a nivel de un prototipo de constructor. Veamos un ejemplo de extensión:

```
/* Definición de una función constructor de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Añadir un método acelerar al prototipo del constructor Coche */
Coche.prototype.acelerar = function()
{
    document.write("acelera<br />");
};

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

/* Llamada al método acelerar del objeto simca1100, accesible a través
del prototipo del constructor Coche */
```



Capítulo 3
Adoptar buenas prácticas

1. Espacio de nombres

1.1 Principio

Cuando desarrollemos, procuraremos no exponer un número excesivo de funciones o variables/constantes 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 un poco como el principio de una caja negra con entradas y salidas bien definidas, pero cuya mecánica interna permanece oculta.

Esto cobra aún más importancia si el código puede utilizarse en otros contextos. El problema es que solemos añadir funciones a nuestros archivos sin tener en cuenta su reutilización. A medida que nuestro proyecto crece, se vuelve cada vez más difícil de mantener. Combinar dos fragmentos de código puede, entonces, provocar conflictos que no resultan evidentes a simple vista y el resultado de la ejecución se vuelve incierto.

Para limitar los conflictos de nombres, añadiremos un contexto adicional (una especie de supercontexto) que garantizará que nuestras funciones y variables/constantes no puedan ser alteradas ni utilizadas accidentalmente. Este espacio de nombres es una especie de contenedor de nombres. Un nombre solo tiene sentido en relación con su espacio de nombres. Por lo tanto, resulta imposible invocar una función que no se encuentre en el espacio de nombres esperado.

Dado que cada código tiene su propio espacio de nombres, ya no existe el riesgo de colisiones asociadas a un código que pertenezca a otro espacio.

Un espacio de nombres es una característica presente en muchos lenguajes. En Java, por ejemplo, la palabra clave `package` se utiliza para realizar la declaración, al igual que `namespace` en C#. En JavaScript, lamentablemente no disponemos de una palabra clave para este fin, pero contamos con otros trucos igualmente potentes para lograrlo.

Este principio es importante para la calidad de sus programas. Una vez que lo haya entendido, se convertirá en algo natural y su forma de desarrollar mejorará aún más.

Cuanto mayor sea su programa, más necesitará espacios de nombres. Es el tamaño de su proyecto lo que impone su uso. Tener en cuenta el espacio de nombres será, por tanto, garantía de un código de mejor calidad, que podrá evolucionar más fácilmente y con mayor seguridad y, por lo tanto, con menos regresiones o efectos secundarios.

1.2 Función

1.2.1 Función interna

Todas las declaraciones realizadas en una función pasan a ser locales de dicha función. De hecho, recordemos que, cuando se declara una variable en una función mediante la palabra clave `let`, su contexto de ejecución está vinculado al de la función y al de un bloque de instrucciones; fuera de este contexto, ya no existe. Por lo tanto, se puede observar que la función es un medio potente para limitar el ámbito de las declaraciones y, por lo tanto, parece un candidato ideal para el espacio de nombres.

Ejemplo

```
function areaRectangulo( longitud, anchura ) {
  const area = longitud * anchura;
  alert( area + " cm²" );
}

areaRectangulo( 10, 5 );
alert( area );   // area no está definida
```

© Editions ENI - All rights reserved

En este ejemplo, la constante `area` se declara dentro de la función `areaRectangulo`, por lo que no existe fuera de ese ámbito; por eso falla la última instrucción `alert( area )`. Nuestra función `areaRectangulo` actúa como un espacio de nombres.

Esta particularidad no se limita únicamente a las declaraciones de variables/constantes, sino que también se aplica a las declaraciones de funciones.

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

  let area = 0;

  function areaRectangulo( longitud, anchura ) {
    const area = ( longitud * anchura );
    return area;
  }
  function areaCuadrado( lado ) {
    return areaRectangulo( lado, lado );
  }
  if ( tipo == "rectangulo" ) {
    area = areaRectangulo( longitud, anchura );
  } else
  if ( tipo == "cuadrado" ) {
    area = areaCuadrado( longitud );
  }
  return area;
}

alert( area( "rectangulo", 10, 20 ) );  // 200
alert( area( "cuadrado", 10 ) );        // 100
alert( areaCuadrado( 20 ) );            // ¡Error!
```

Las funciones `areaRectangulo` y `areaCuadrado` no existen fuera de la función `area`, como demuestra el hecho de que nuestra última instrucción, que consistía en utilizar la función `areaCuadrado`, provocó un error de ejecución.

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

1.2.2 Función anónima

Si queremos proteger la ejecución de nuestro código, también es posible utilizar una función anónima (o función flecha), como vimos brevemente en el primer capítulo. Esta función sin nombre garantizará un contexto independiente.

Ejemplo

```
(function () {
  function areaRectangulo(longitud, anchura) {
    const area = ( longitud * 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 en cuanto 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.

Por lo tanto, una vez finalizada la ejecución, todo lo que se encontraba dentro de esta función anónima queda inaccesible.

La función anónima sirve, por tanto, como espacio de nombres para nuestro código. Esto explica por qué falla la invocación posterior de la función `areaRectangulo`.

La ejecución es segura, ya que no podemos crear colisiones en nuestros nombres de funciones. Si asociamos nuestro código a otro código que, a su vez, ha definido la función `areaRectangulo`, nuestro código seguirá funcionando con normalidad.

© Editions ENI - All rights reserved

Cabe señalar que la sintaxis de la función flecha es más compacta. En el ejemplo siguiente, para mostrarle que nuestra función flecha no altera lo que ya está en el espacio global, hemos escrito una nueva función `areaRectangulo` antes de invocar nuestra función flecha.

```
function areaRectangulo( longitud, anchura ) {
    alert( `El área del rectángulo es ${longitud} * ${anchura}` );
}

(() => {
    function areaRectangulo( longitud, anchura ) {
        const area = ( longitud * anchura );
        return area;
    }
    function areaCuadrado( lado ) {
        return areaRectangulo( lado, lado );
    }
    alert( areaRectangulo( 10, 20 ) );
    alert( areaCuadrado( 20 ) );
} )();


areaRectangulo( 100,200 ); // El área del rectángulo es 100x200
```

Aunque nuestra función flecha también contiene una función `areaRectangulo`, esta no interfiere con la que se había definido anteriormente, la cual sigue funcionando con normalidad en la última llamada.

1.2.3 Función anónima con parámetros

El caso anterior es sencillo, pero insuficiente en la práctica, ya que nos gustaría disponer de una función `area` accesible desde cualquier lugar, por ejemplo, sin que las funciones auxiliares `areaCuadrado` y `areaRectangulo` sean visibles.

Para ello, podemos crear un objeto vacío que nos servirá de contenedor para las funciones accesibles. Este actuará entonces como espacio de nombres para ellas.

Ejemplo

```
miArea = {};

(function ( ns ) {
  function areaRectangulo( longitud, anchura ) {
    const area = ( longitud * 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 de área imposible";
  }

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

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

En este ejemplo, hemos creado una función area interna a una función anónima. Esta utiliza las funciones `areaCuadrado` y `areaRectangulo`. Para que la función `area` se pueda utilizar desde cualquier lugar, hemos hecho que la función anónima que se ejecuta reciba como parámetro un objeto que almacena la referencia al método `area`.

Este objeto actuará como espacio de nombres para el método `area`. Las otras funciones `areaCuadrado` y `areaRectangulo` quedan definitivamente ocultas para el usuario.

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

```
ns.area = ns.area ?? area
```

© Editions ENI - All rights reserved

De este modo, con este sistema obtenemos algo similar a los métodos privados y públicos en la programación orientada a objetos. La función anónima desempeña entonces el papel de clase.

El hecho de que el método `area` pueda seguir funcionando fuera de la función anónima está relacionado con el principio de clausura. La clausura es la capacidad que tiene una función accesible de utilizar un contexto al que ya no se puede acceder, por lo que ese contexto es privado.

La variable `miArea` se encuentra 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 que sus propiedades son visibles en el espacio de nombres global. Al sustituir `miArea` por `window.miArea`, reducimos la visibilidad de nuestra variable `miArea`, que pasa a ser una propiedad de `window`. En la práctica, no hay realmente ninguna diferencia.

En general, añadir nuevas funcionalidades a `window` depende bastante del contexto web; se buscará aportar capacidades adicionales a la ventana.

Cabe señalar que las colisiones de nombres están limitadas por los espacios de nombres, pero siempre que estos también sean distintos. De lo contrario, el problema se traslada a un nivel superior. Por lo general, se utilizará una convención para la elección de un nombre de espacio de nombres de tipo URI (*Uniform Resource Identifier*) para asegurarse de ser los únicos propietarios. Por ejemplo, si trabajamos en una empresa ABC, podríamos hacer que el objeto `miArea` se llamara `abc`, o incluso `com_abc`...

1.3 Clausura

Ya hemos utilizado una clausura para nuestra función `area`, pero, para ser más explícitos, aquí vamos a usarla de otra manera, conservando al final solo una función de acceso.

```
const miArea = ( function () {
  function areaRectangulo( longitud, anchura ) {
    const area = ( longitud * 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 de área imposible";
  }
  return function() {
    return area.apply(this, arguments);
  };
} )();

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

En este ejemplo, nuestra función anónima devuelve una nueva función. Es esta nueva función la que conserva la lógica de funcionamiento que queremos exponer. Aquí, hacemos que la función `area`, invisible, pueda utilizarse con cualquier número de argumentos; por eso empleamos `apply`.

La clausura funciona correctamente, ya que la función devuelta sigue accediendo a las funciones `area`, `areaCuadrado` y `areaRectangulo`, mientras que estas últimas son inaccesibles en el contexto global.

Esta técnica no es adecuada si tiene varias funciones a las que quiere dar acceso.

1.4 Clase

Como acabamos de ver, el espacio de nombres que genera la función anónima sirve para declarar funciones o variables cuyo uso es interno y, por lo tanto, invisible fuera de la función. Este sistema es similar al que podemos encontrar en la programación orientada a objetos con métodos de clase de acceso público o privado. Gracias a ello, podremos crear una nueva forma de clase que combine a la vez una parte oculta con una parte accesible.

© Editions ENI - All rights reserved

Volvamos a nuestro ejemplo de cálculo de área. Tenemos:

```
const figura = ( function () {
  // Parte privada

  function areaRectangulo( longitud, anchura ) {
    const area = ( longitud * 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 de área imposible";
  }

  // Parte pública de nuestro objeto

  return {
    longitud: 0,
    anchura: 0,
    setLongitud: function( longitud ) {
      this.longitud = longitud;
    },
    setAnchura: function( anchura ) {
      this.anchura = anchura;
    },
    area: function() {
      alert( area( this.longitud, this.anchura ) );
    }
  }
} )();

figura.setLongitud( 10 );
figura.setAnchura( 10 );
figura.area();
figura.areaCuadrado( 10 ); // ¡Imposible!, porque es privada
```