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 );
```

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`.

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.

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 ) );
```