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
```

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.

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
```

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.

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
```