Distintas formas de conseguir privacidad en JavaScript

Una de las cosas que echa en falta la mayoría de programadores cuando empieza a desarrollar en JavaScript es la visibilidad de las funciones, propiedades o variables. En efecto **JavaScript no tiene modificadores de visibilidad** y todo es, por defecto, accesible (vamos, lo que en la mayoría de lenguajes se conoce como público).

Que no exista un modificador para especificar un modificador de visibilidad no significa que **no puedan declararse miembros privados**. Se puede y de hecho es (por supuesto) una buena práctica. Veamos cómo podríamos hacerlo de distintas maneras.

Opción 1 – No hacer nada y establecer nomenclatura

No es una opción real pero **es muy utilizada hoy en día**. Mucha gente usa el guión bajo (o a veces el doble guión bajo) para indicar que algo es *privado* y que no debe ser modificado directamente. **Por supuesto, eso es una convención**, nada impide realmente modificar el valor del miembro afectado:

```
var Beer = function(name) {
    this.__name = name || "unnamed";
    this.setName = function(newname) {
        if (newname) {
            this.__name = newname;
        }
    }
    this.getName = function() {
        return this.__name;
    }
}

var beer = new Beer("Punk IPA");
console.log(beer.getName());        // Imprime Punk IPA
beer.setName();
console.log(beer.getName());        // Imprime Punk IPA
beer.__name = undefined;
console.log(beer.getName());        // Imprime undefined
```

Se puede ver como, a pesar de que la función setName evita que se pueda establecer el nombre a undefined, nada impide que se pueda acceder internamente a la variable __name. Aunque, como vamos a ver ahora, existen mecanismos para crear realmente variables (y funciones) privadas.

Opción 2 – Retornar solo la parte pública

Por defecto si no devolvemos nada, las funciones en JavaScript devuelven undefined, excepto si son llamadas con new que entonces devuelven el propio objeto que se está creando (y que se referencia por this). Pero nada impide devolver *otro* objeto que contenga tan solo los miembros públicos:

```
var Beer = function(name) {
    var __name = name || "unnamed";
    var setName = function(newname) {
        if (newname) {
            __name = newname;
        }
    }
    var getName = function() {
        return __name;
    }

    return {
        getName: getName,
        setName: setName
    }
}

var beer = new Beer("Punk IPA");
var beer2 = new Beer("Hoegaarden");
beer2.__name = "Sang de rabo de drac";
beer2.setName("Devil's IPA");
console.log(beer.getName());    // Imprime Punk IPA
console.log(beer2.getName());   // Imprime Devil's IPA
console.log(beer.__name);       // Imprime undefined
console.log(beer2.__name);      // Imprime Sang de rabo de drac
```

Observa ahora que **dentro de la función Beer no usamos this para nada**. Creamos tres miembros (__name, setName y getName) y luego devolvemos un objeto que contiene solo los dos miembros públicos. Fíjate que si establecemos desde fuera el valor de __name, **lo que en realidad estamos haciendo es crear un nuevo miembro __name dentro del objeto,** miembro que nada tiene que ver con el __name que está dentro de la función Beer. Así, puedes observar como beer2.getName() devuelve "*Devil's IPA*" (que es el valor establecido con setName) y no "*Sang de rabo de drac*" que es el valor del miembro __name creado por nosotros. **A todos los efectos la variable __name es inaccesible desde fuera del código de la función Beer** (observa también que beer.__name vale *undefined*, ya que a todos los efectos el objeto beer no tiene ningún miembro __name).

Ah, por cierto: si usas esa técnica (devolver solo la parte pública) **no es necesario que los objetos se creen con new**. En este caso *beer = new Beer();* es equivalente a *beer = Beer();* ya que new lo único que hace es que en la función llamada (Beer) el valor de this referencie al nuevo objeto que se

está creando y que se devuelva this por defecto (y no undefined). Pero si, como es este caso, no usas this dentro de la función y devuelves algo de forma explícita, new pasa a ser totalmente opcional. **Sí: JavaScript es muy distinto a Java o a C#**.

Opción 3 – Funciones adicionales fuera de `this`

```
var Beer = function(name) {
    var __name = name || "unnamed";
    this.setName = function(newname) {
        if (newname) {
            __name = newname;
        }
    }
    this.getName = function() {
        return __name;
    }

    this.dump = function() {
        console.log("Beer is " + prettyName());
    }

    var prettyName = function() {
        return __name.toUpperCase();
    }
}

var beer = new Beer("Punk IPA");
beer.dump(); //Beer is PUNK IPA
```

Si por el contrario en la función prettyName en vez de utilizar __name llamamos al método this.getName() se producirá un error.

```
var Beer = function(name) {
    var __name = name || "unnamed";
    this.setName = function(newname) {
        if (newname) {
            __name = newname;
        }
    }
    this.getName = function() {
        return __name;
    }

    this.dump = function() {
        console.log("Beer is " + prettyName());
    }

    var prettyName = function() {
        //return __name.toUpperCase();
        return this.getName().toUpperCase();
    }
}

var beer = new Beer("Punk IPA");
beer.dump(); //error
```

Cuando desde una función privada llamamos a un método público tenemos un error. Eso **es debido a que el método privado prettyName no forma, realmente, parte del objeto**(nunca lo hemos asignado a this) y por lo tanto, dentro de este método el valor de this no es el del objeto, si no que es el del contexto global (window en un navegador).

¿Cuál es la solución a este problema? hay dos.

La primera **consiste en guardarse el valor de this dentro de *otra* variable y usar dicha variable para acceder al objeto**. Comúnmente se llama self o that a esa variable, aunque eso es simplemente una convención:

```
var Beer = function(name) {
    var __name = name || "unnamed";
    //guardamos el contexto que tiene en este momento this para poder usarlo posteriormente
    var self = this;
    this.setName = function(newname) {
        if (newname) {
            __name = newname;
        }
    }
    this.getName = function() {
        return __name;
    }

    this.dump = function() {
        console.log("Beer is " + prettyName());
    }

    var prettyName = function() {
        //el contexto de this en este momento es window
        //por eso utilizamos self
        return self.getName().toUpperCase();
    }
}

var beer = new Beer("Punk IPA");
beer.dump();
```

El segundo método es más elegante porque no requiere usar ninguna variable extra: consiste en usar *call* para forzar que dentro de prettyName la variable this tome el valor que queremos; esto es, el del propio objeto.

```
var Beer = function(name) {
    var __name = name || "unnamed";
    this.setName = function(newname) {
        if (newname) {
            __name = newname;
        }
    }
    this.getName = function() {
        return __name;
    }

    this.dump = function() {
        //forzamos a que el contexto de this sea el propio objeto
        console.log("Beer is " + prettyName.call(this));
    }

    var prettyName = function() {
        return this.getName().toUpperCase();
    }
}

var beer = new Beer("Punk IPA");
beer.dump(); //Beer is PUNK IPA
```

Opción 4 – Utilizar un WeakMap (ES6)

Se puede utilizar un WeakMap para evitar el rendimiento y la penalización de memoria del enfoque anterior. WeakMaps asocia datos con Objetos (aquí, instancias) de tal manera que solo se puede acceder usando ese Mapa Débil.

Por lo tanto, usamos el método de variables de ámbito para crear un WeakMap privado, luego usamos ese WeakMap para recuperar datos privados asociados con this . Esto es más rápido que el método de variables de ámbito porque todas las instancias pueden compartir un único WeakMap, por lo que no es necesario que vuelva a crear métodos solo para hacer que accedan a sus propios WeakMaps.

```
let Beer = (function(name) {

    let privateProps = new WeakMap();

    class Beer{

        constructor(name){
            this.name = name; // name is public
            privateProps.set(this, {otherProp:'xxx'}); // otherProp is private
        }

        greet(){
            console.log(`name: ${this.name}, otherProp: ${privateProps.get(this).otherProp}`);
        }
    }
return Beer;

})();

var beer = new Beer("Punk IPA");
beer.greet(); //name: Punk IPA, otherProp: xxx
```