UT04.5: Patrones de Objetos en ES5

Contenido

1. Introducción 2
2. Compartir campos privados en una herencia 2
   2.1. Invocar métodos sobrescritos de la clase padre 3
3. Patrón Factory 4

1. Introducción

En este documento vamos a estudiar cómo podemos encapsular información en campos privados, por ejemplos arrays o referencias a objetos, y que puedan ser heredados en las subclases, es decir, lo métodos de las clases padres puedan acceder a los campos privados, pero también los métodos de las subclases puedan **compartir** la misma referencia.

Por último, vamos a implementar un patrón que nos permite crear objetos sin tener disponible la función constructora, es lo que denominamos un patrón **Factory**.

2. Compartir campos privados en una herencia

El problema que tenemos al tener una estructura de herencia es que tanto en la clase padre, como en la clase hija debemos apuntar a la misma referencia para que los métodos específicos de la clase hija puedan hacer uso del mismo objeto.

Como indicamos en la introducción, la idea es *compartir* la referencia creándola en las clases hijas, y pasarla a las clases padre utilizando el constructor.

Veamos un ejemplo de implementación de listas. El siguiente código muestra la implementación de una clase *List* aunque no tiene implementados todos los métodos. Como parámetro en la función constructora hemos definido *list* que tiene como valor por defecto un array vacío que contendrá los objetos de la lista.

```
function List(capacity = 10, list = []){
    let _capacity = capacity;
    let _list = list;

    this.add = function (elem) {
        if (this.isFull()) { //Lanzamos excepción si la lista está llena
            throw new Error ("The list is Full. You can't put the element
in it");
        }
        _list.push(elem);
        return this.size(); //Devolvemos el tamaño.
    }
}
```

Ahora implementamos una subclase *ObjectList* con la misma estrategia que la clase padre respecto a los parámetros de la función constructora. En el momento de invocar al superconstructor debemos pasar como argumento la referencia creada en la subclase, con lo que estaremos compartiendo el mismo objeto entre ambas clases.

```
function ObjectList(capacity = 10, type, list = []){
    if (!(type instanceof Function)) throw new Error ("The list doesn't
have a specific type");

    let _list = list;
    List.call(this, capacity, _list);
    let _type = type;
}
ObjectList.prototype = Object.create(List.prototype);
ObjectList.prototype.constructor = ObjectList;
```

2.1. Invocar métodos sobrescritos de la clase padre

En la especialización de una subclase nos puede interesar mantener la funcionalidad de métodos de la clase padre, pero especializando el comportamiento de la clase hija. El problema que se nos presenta es que la clase hija sobrescribe el acceso al método al asignarlo a la nueva función.

La solución en este caso pasa por guardar la referencia al método en una variable privada, e invocarlo pasando el contexto de la subclase en una invocación al método `call()`.

```
let addList = this.add; // Método heredado salvado en una variable
  this.add = function (elem) {
  if (!(elem instanceof _type)) { //Lanzamos excepción si el elemento no
es del tipo
      throw new Error ("The element is not a " + _type.name);
  }
  return addList.call(this, elem); // Invocación del método heredado
pasándole nuestro contexto.
}
```

3. Patrón Factory

Hay un detalle evidente en el diseño anterior y es que podemos crear un objeto *List* o un objeto *ObjectList* pasando el array, con lo que tendríamos la referencia desde el exterior pudiéndonos saltar la lógica de negocio de ambas clases. La solución pasa por implementar un patrón *Factory* que proteja la creación de los objetos de tipo lista.

El patrón de diseño Factory (o patrón de fábrica) es un patrón de diseño creacional que proporciona una interfaz para crear objetos controlado cómo se crean, e impone las restricciones que deseemos.

El patrón Factory también lo podemos utilizar cuando una clase no puede anticipar la clase de objetos que debe crear. En lugar de llamar directamente al constructor de una clase para crear un objeto, se utiliza una interfaz común (la fábrica) para la creación de objetos. Esta interfaz puede ser una interfaz o una clase abstracta que define un método para crear objetos.

Este patrón proporciona flexibilidad al código y facilita la creación de familias de objetos relacionados sin especificar sus clases concretas. Veamos un ejemplo.

Creamos una función *ListFactory()*, se trata de una función constructora del patrón *factory*. En el interior creamos la jerarquía de clases que queremos compartir para que no puedan instanciarse objetos desde fuera de la función constructora. Por último, implementamos métodos *créate()* que permitan crear los objetos deseados, estos métodos reciben los mismos argumentos que los constructores de las clases.

```
function ListFactory(){
    function List(capacity = 10, list = []){
        ...
    }

    function ObjectList(capacity = 10, type, list = []){
        ...
    }
    ObjectList.prototype = Object.create(List.prototype);
    ObjectList.prototype.constructor = ObjectList;

    this.createList = function(capacity = 10){
        return new List(capacity);
    }
    this.createObjectList = function(capacity = 10, type){
        return new ObjectList(capacity, type);
    }
}
```