UT04.5: Patrones de Objetos en ES5

Contenido

1. Introducción

De manera breve, podemos definirlos como una serie de **técnicas** que permiten **organizar nuestro código** para resolver **problemas comunes** que nos ocurren a todos los desarrolladores de software. Se tratan de soluciones de problemas mediante diseño de código, las cuales han sido ya probadas y testeadas para verificar su comportamiento con anterioridad. Además, proporcionan un vocabulario común para los desarrolladores.

Hoy en día es uso de estos patrones de diseño es fundamental en desarrollo de software y por lo tanto son un requisito que todo programador tiene que conocer. Existen tres tipos de categorías de patrones:

- **Patrones Creacionales**: Son los patrones que nos solucionan nuestros problemas a la hora de crear instancias.
- **Patrones de Comportamiento**: Son los patrones de diseño que se encargan de definir las formas en las que interactúan y reparten responsabilidades las distingas clases y objetos.
- **Patrones Estructurales**: Son los patrones de diseño que tratan la composición de las clases y objetos.

Un ejemplo de patrón es *Prototype* en base al cual se organizan objetos en JavaScript. En este apartado vamos a centrarnos en algunos patrones que podemos utilizar en JavaScript de tipo creacional.

2. Patrón IIFE (Immediately invoked function expression)

De este patrón ya hemos hablado anteriormente. Este patrón genera una expresión que permite invocar una función anónima en el momento de su definición, por lo que garantizamos que el código de la función anónima se ejecute una única vez. Este patrón es la base para construir otros tipos de patrones en JavaScript.

2.1. Esquema del patrón

Lo habitual es que IIFE devuelva un objeto que asociemos a una variable para poder utilizarlo de forma global en nuestra aplicación. La función anónima debe estar contenida entra paréntesis para que pueda ser invocada, y el resultado es asociado a una variable, en el ejemplo *app*.

En el cuerpo de la función declaramos variables y funciones privadas, las cuales tiene ámbito local por definición. El valor de retorno es un objeto con la función privada como propiedad, por lo que tenemos un *closure* que da acceso a los campos privados de la función anónima.

```
const app = (function () { // Función principal anónima
  const carId = 123; // Variable con ámbito local solo accesible desde la
función anónima y por tanto encapsulada.
  const getId = function () { // función que permite cambiar el contenido
de la variable local.
    return carId;
  };
  return { // Objeto devuelto por función anónima y asignado a app.
    getId, // propiedad del objeto que hace referencia a la función
local.
  };
}()); // Invocación de la función anónima. Se hace justo en el momento de
su declaración.
// El objeto global puede acceder al ámbito de la función anónima a
través de sus funciones internas.
$$result.log(app.getId()); // 123
```

2.2. Ejemplo de implementación

Queremos crear una clase *Book* para una aplicación de una librería. Para cada libro necesitamos una propiedad *title* e *isbn* verificada por medio de una función. Además, necesitamos llevar un control del número total de objetos *Book* que hemos instanciado.

Básicamente necesitamos una función constructora que sea además un *closure*. Para ello utilizamos el patrón IIFE para asociar a una variable *Book*, el retorno de la función anónima, el cual debe ser la función constructora para *Book*. Como campos privados tenemos la variable *numOfBooks* que nos servirá de constructor, y una función privada *checkIsbn()* que comprueba el formato del isbn del libro, sino lanza una excepción de tipo erróneo.

El constructor *Book* tiene acceso a los campos privados por estar en el mismo ámbito. Añadimos un método al prototipo del constructor *getNumOfBooks()* que también tendrá acceso a los campos privados.

Por último, la función anónima debe devolver la función constructora, la cual funcionará como un *closure* fuera la función anónima.

```
const Book = (function () { // Utilizamos una función anónima.
  // private static field
  // Esta variable es solo accesible desde dentro del objeto.
  let numOfBooks = 0;

  // private static method
  // Este método es solo accesible desde el objeto.
  function checkIsbn(isbn) {
    if (!(/^(\d{3}-\d{2}-\d{5}-\d{2}-\d)$/.test(isbn)
      || /^(\d-\d{6}-\d\d-\d)$/.test(isbn))) { throw new TypeError('isbn
is not valid!'); }
  }
  // Construcctor de objetos Book
  function Book(isbn, title) {
    checkIsbn(isbn); // Validamos isbn
    this.isbn = isbn; // Asignamos el isbn
    this.title = title; // Asignamos el title
    ++numOfBooks; // Incrementamos el contador. El valor del contador se
mantiene porque es un closure
  }
  // Método que devuelve el número de libros creados. Será heredado por
todas las instancias creadas.
  Book.prototype.getNumOfBooks = function () {
    return numOfBooks; // Este método hace accesible numOfBooks.
  };

  return Book; // Devolvemos la función constructora.
}()); // Se invoca la función anónima para obtener el constructor Book.
```

La variable *Book* contiene la función constructora, por lo que la podemos invocar con el operador new. El siguiente código testea la función constructora.

```
const firstBook = new Book('012-94-33296-04-2', 'First Title');
$$result.log(firstBook.title); // First Title
$$result.log(firstBook.getNumOfBooks()); // 1
const secondBook = new Book('0-943396-04-2', 'Second Title');
$$result.log(firstBook.title); // First Title
$$result.log(secondBook.title); // Second Title
$$result.log(firstBook.getNumOfBooks()); // 2
$$result.log(secondBook.getNumOfBooks()); // 2
```

Además del *closure*, este esquema tiene la ventaja de que *checkIsbn()* no está creado en cada tipo de instancia de *Book*, lo que si que estaría si hubiéramos incluido en el cuerpo del constructor.

3. Factoría de objetos abstractos

Una clase abstracta es aquella que permite heredar sus propiedades y métodos, pero no podemos instanciar objetos de ella porque la implementación de sus métodos es incompleta. Las clases que heredan de la clase abstracta son las que completan la implementación para poder instanciar objetos.

Vamos a crear una jerarquía de formularios, siendo *BaseForm* la raíz de la jerarquía, pero no queremos instanciar objetos de ella, solamente permitir que se hereden ciertas propiedades, y en las clases hijas se implementen sus métodos. Esta clase será *GridForm*.

Implementamos el patrón por partes. Utilizando la expresión IIFE vamos a definir ambos constructores, y utilizamos un variable que sirve de cerrojo para evitar que el constructor *BaseForm* pueda instanciar objetos.

```
(function(){
  let abstractCreateLock = false; //Definición del cerrojo.

})(); //Invocamos la función global.
```

Para impedir que *BaseForm* pueda instanciar objetos comprobamos que el cerrojo. Si es *true* lanzamos una excepción para que no podamos seguir con la ejecución del constructor, por lo que para poder continuar necesitamos un *false* en el cerrojo.

Justo después de la comprobación del cerrojo debemos volver a activarlo para impedir la instancia de objetos.

Añadimos dos métodos uno con implementación y otro sin ella.

```
function BaseForm(){
  if(abstractCreateLock)
    throw new Error("You can't instantiate BaseForm!");
  abstractCreateLock = true; //Reactivamos el seguro.
  this.property = "default"; //Propiedad heredable por la subclase
}
BaseForm.prototype = {};
//Método sin implementación. Se debe implementar en la subclase
BaseForm.prototype.post = function(){
  throw new Error("Not implemented!");
}
//Método con implementación. Se puede reutilizar en la subclase.
BaseForm.prototype.get = function(){
  return ("Get Operation!!!!!!!");
}
```

La clase *GridForm*, como subclase de *BaseForm* debe llamar al constructor, pero antes de hacerlo tenemos que desactivar el cerrojo que nos permita ejecutar el constructor completo. Una vez ejecutado el superconstructor podríamos añadir propiedades específicas para *GridForm*.

Como métodos, al menos debemos añadir el método *post()* para darle implementación en la subclase, el resto pueden ser heredados.

```
// Definimos la subclase GridForm que hereda de BaseForm
function GridForm(){
  abstractCreateLock = false;//Abrimos el cerrojo e invocamos BaseForm
  BaseForm.call(this); //Heredamos las propiedades de la superclase.
}
GridForm.prototype = Object.create(BaseForm.prototype);
GridForm.prototype.post = function(){ //Implementamos el método
  // ...
  return "Grid is posted.";
}
```

Por último, debemos activar el cerrojo antes de finalizar la ejecución de la función anónima y resolver el problema de retornar don funciones al mismo tiempo que puedan acceder a la variable del cerrojo.

El objeto `window` representa la ventana del navegador donde hemos cargado la página. Todas las funciones y variables globales son propiedades de este objeto, por lo que si le añadimos dos propiedades y le asignamos y le asignamos a cada una un constructor de nuestra estructura, estaremos habilitando estas funciones de forma global, simulando el retorno de las dos funciones.

```
abstractCreateLock = true; //Activamos el cerrojo.
window.BaseForm = BaseForm;
window.GridForm = GridForm;
```

Con el siguiente código podemos ver como una instancia de *GridForm* tiene implementado los métodos *post()*, de forma local, y *get()* heredado de *BaseForm*.

Al intentar instanciar *BaseForm* el cerrojo lo impide, y enviamos una excepción.

```
const myGrid = new GridForm(); // Instanciamos un objeto GridForm
$$result.log(myGrid.post()); // Grid is posted.
$$result.log(myGrid.get()); // Get Operation!!!!!!!
$$result.log(myGrid.property); // default
try {
  $$result.logBold('Instanciando objeto de clase abstracta');
  const myForm = new BaseForm(); // Al intentar instaciar se genera la
excepción.
} catch (err) {
  $$result.log(err.message); // Error: Can’t instantiate BaseForm!
}
$$result.logBold('Comparando tipos GridForm con BaseForm');
$$result.log(myGrid instanceof BaseForm); // true. myGrid es una
instancia de BaseForm.
```

4. Interfaz

Un interfaz es realmente una clase abstracta en la que todos sus métodos están sin implementar. En JavaScript no podemos heredar de un interfaz, pero podemos implementar una estructura para comprobar que están todos los métodos implementados en la clase.

Primero implementamos la clase *Interface* que tiene como propiedades *name*, con el nombre del interfaz, y *methods* con un array con los nombres de los métodos que deben ser implementados. El array es copiado a la propiedad para no perder los valores por ser una referencia.

```
let Interface = function (name, methods) {
  this.name = name; //Nombre del Interfaz
  // copies array
  this.methods = methods.slice(0); //Array con el listado de métodos.
};
```

A la función constructora le asignamos un método estático *Interface.checkImplements()*, el cual recibe la referencia del objeto y el interfaz que debe implementar. El método itera sobre el array con los métodos chequeando si el objeto tiene la propiedad implementada, y se trata de una función. En caso de encontrar que una propiedad no está implementada lanza una excepción.

```
Interface.checkImplements = function (obj, interfaceObj) {
  for (let method of interfaceObj.methods) { //Recorremos el array
    if (!obj[method] || typeof obj[method] !== "function"){
      // Si no está implementado el método se lanza excepción.
      throw new Error("Interfacenotimplemented! Interface: " + interfaceO
bj.name + " Method: " + method);
    }
  }
};
```

Para testear la clase *Interface*, definimos un interfaz *IMaterial*, con el conjunto de métodos a implementar. Declaramos una clase *Product* cuya primera instrucción es chequear el interfaz. Si todos los métodos están implementados, el método nos permitirá instanciar el objeto *Product*. En caso contrario, se lanza la excepción impidiendo instanciar el objeto.

```
let iMaterial = new Interface("IMaterial", ["getName", "getPrice"]);
function Product(name,price){
  Interface.checkImplements(this, iMaterial);
  this.name = name;
  this.price = price;
}
Product.prototype.getName = function(){
  return this.name;
};
Product.prototype.getPrice = function(){
  return this.price;
};

const p1 = new Product('HP EliteBook', 1500);
$$result.log(`${p1.getName()} ${p1.price}`); // HP EliteBook 1500
```

```
// Borramos uno de los métodos de Product
delete Product.prototype.getPrice;
// Intentamos instanciar un nuevo Product.
try {
  // No se puede instanciar porque falta un método del interfaz y se
genera la excepción.
  const p2 = new Product('HP Pavilion', 1500);
} catch (err) {
  $$result.log(err.message); // Error: Interface not implemented!
}
```

Con esta solución no podemos utilizar el operador `instanceof` porque el operador hace una comprobación a base de prototipos, y el objeto tiene los métodos implementados, pero no tiene asociado el su prototipo el prototipo del interfaz.

5. Patrón Singleton

Una necesidad muy habitual de las aplicaciones es crear una instancia de un tipo de objeto, pero solamente una, y que se pueda reutilizar en diferentes partes del código. Esta instancia no queremos que esté al alcance desde una variable local. Lo podemos logar implementando este patrón.

5.1. Esquema del patrón

Utilizamos una expresión IIFE para crear un *closure* que devuelva el objeto único. La función anónima tiene tres partes que pasamos a revisar.

En primer lugar, tenemos un campo privado que almacenará la instancia con el objeto que queremos crear. Este campo lo hemos denominado *instantiated*.

```
let Singleton = (function () {
  let instantiated; //Campo privado para guardar la instancia única.
})();
```

En segundo lugar, necesitamos un método privado a la función anónima, la denominamos *init()*, encargada de generar el objeto único que necesitamos. El método debe ejecutarse una única vez. Como vemos en este ejemplo, *init()* funciona como un constructor donde estamos definiendo métodos y campos privados, y devuelve el objeto que será único. En este caso, tenemos un objeto literal con las propiedades y los métodos públicos del objeto único.

```
function init() {
    // Métodos y campos privados
    function privateMethod(){
        console.log("I am private");
    }
    let privateVariable = "Im also private";
    let privateRandomNumber = Math.random();
  return { // Devuelve el objeto que será único.
    publicMethod: function () {
      return 'publicMethod unique instance';
    },
    publicProperty: 'publicProperty unique instance',
    getRandomNumber: function() {
      return privateRandomNumber;
    }
  };
}
```

Por último, la función anónima tiene que crear la instancia del objeto único, y devolver un objeto que nos de acceso a la instancia única. Como vemos, estamos devolviendo un objeto literal con un único método *getInstance()*. Este método, si no tiene nada asignado al campo privado invoca al método privado de la función anónima *init()* para crearlo y devolverlo. Si ya está creado, directamente devuelve la instancia única. La instancia única queda almacenada en un *closure*, y garantizamos que solo se haya instanciado una única vez.

```
  return {
    // Devuelve un objeto con el método getInstance
    getInstance: function () {
      if (!instantiated) { //Si instantiated es undefined llama a init.
        instantiated = init(); //instantiated contiene el objeto único
      }
      return instantiated; //Devuelve el campo privado
    }
  };
```

Para testear el patrón debemos invocar desde el objeto *Singleton* al método *getInstance()*, el cual siempre devolverá la misma instancia.

```
$$result.log(Singleton.getInstance().publicMethod()); // publicMethod
unique instance
$$result.log(Singleton.getInstance().publicProperty); // publicProperty
unique instance
$$result.log(Singleton.getInstance().getRandomNumber()); // Random
number.
$$result.logBold('Comparando instancias');
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();
// Chequeamos si las dos instancias son el mismo objeto.
$$result.log(instance1 === instance2); // true
```

5.2. Alternativas de implementación

Vamos a crear un sistema para generar *logs* en una aplicación utilizando un objeto único. El primer método utilizado es a través de un objeto literal. Tenemos una propiedad *enabled* que indica si queremos activar la funcionalidad o no. Si la propiedad no está activa impedimos la ejecución, aunque también podríamos lanzar una excepción.

```
//Declaración mediante objeto literal
let Logger1 = {
  enabled:true, //La propieda dice si se puede ejecutar el método o no.
  log: function(logText){ //Asociamos un método al objeto
    if(!this.enabled) // Comprobamos si la propiedad está activa
      return; // Si no está activa salimos del método.
    if(console && console.log) // Ejecutamos el método.
      console.log(logText);
    else
      alert(logText);
  }
}
```

Esta solución puede ser válida, aunque siempre podemos hacer una copia del objeto. El segundo método sería con una función constructora, pero impedimos la creación de objetos mediante una excepción. La propiedad *enabled* y el método *log()* los asociamos estáticamente a la función constructora.

```
function Logger2(){ //Función constructora.
  throw new Error("You can't create a new Logger object.");
}
Logger2.enabled = true; //La propieda enabled
Logger2.log = function(logText){
  if(!Logger2.enabled) //
    return;
    if(console && console.log)
      console.log(logText);
    else
      alert(logText);
};
```

Para testear el código invocamos a los objetos creados.

```
Logger1.log('Logger1: test'); // Logger1: test
Logger1.enabled = false;
Logger1.log('Logger1: test'); //
Logger2.log('Logger2: test'); // Logger2: test
Logger2.enabled = false;
Logger2.log('Logger2: test'); //
```

6. Patrón Módulo

Este patrón nos permite "empaquetar" funcionalidad para ser utiliza desde el objeto que contiene el módulo.

La base de este patrón es muy similar al Singleton. Utilizamos una expresión IIFE para contener la funcionalidad que necesitamos, pero el objeto que devuelve la función anónima ya contiene todas las propiedades y métodos públicos que necesitemos en nuestra aplicación. Estas propiedades y métodos tendrán acceso a los campos y métodos privados de la función anónima.

```
let someModule = (function () { //La función anónima devuelve el módulo
  // Campos privados. Implementan la funcionalidad del módulo
  let privateVar = 5;
  // Métodos privados
  let privateMethod = function () {
    return 'Private Test';
  };
  return { //Interfaz pública del módulo
    // Propiedades públicas
    publicVar: 10,
    // Métodos públicos
    publicMethod: function () {
      return ' Followed By Public Test ';
    },
    // Métodos que dan acceso a datos privados.
    getData: function () {
      return privateMethod() + this.publicMethod() + privateVar;
    }
  }
})(); // Invocamos la función anónima.
```

En el caso de querer ampliar la funcionalidad del módulo podemos hacerlo nuevamente con una expresión IIFE. A la función anónima le pasamos el módulo, y así podemos añadirle como propiedades la funcionalidad que necesitemos. El módulo es pasado en la invocación, y tenemos la precaución de comprobar si existe o no con una expresión de cortocircuito.

```
//Añadimos funciones extra al módulo.
//Podemos añadir nueva funcionalidad el módulo para ampliar su uso.
someModule = (function (someModule) {
    someModule.extension = function () {//Declaramos nueva funcionalidad.
        return "another method!";
    };
    return someModule; //Devolvemos el objeto con la nueva funcionalidad.
})(someModule || {}); //Precaución por si someModule no existe.
```

Comprobamos el funcionamiento del módulo.

```
$$result.log(someModule.getData()); // Private Test Followed By Public
Test 5
$$result.log(someModule.extension()); // another method!
```

7. Patrón Iterador

Este es el último patrón que vamos a implementar. Pensemos en el siguiente contexto, queremos recorrer los valores almacenados en una colección, básicamente un array, aunque podría ser cualquier tipo de objeto. Lo habitual es utilizar un bucle para recorrer todos los elementos, el problema es si queremos detener momentáneamente la ejecución del bucle en un determinado valor u objeto de la colección, para luego retomar la colección en el mismo punto anterior.

Romper la ejecución del bucle lo podemos hacer con un `break`, pero necesitaríamos lógica extra para guardar el índice del último elemento con el que hemos trabajado, lo que complicar mucho el código. El siguiente código muestra cómo romper un bucle.

```
for (let i = 0; i < 10; i++){
  if (i > 2) break;
  console.log(i * 2);
}

[1, 2, 3, 4, 5].forEach (value => {
  if (i > 2) break;
  console.log (value * 2);
});
```

Los **objetos iteradores** nos permiten detener la recuperación de los elementos de una colección, para proseguir cuando lo necesitemos en el mismo punto donde lo habíamos dejado. Por definición, un iterador es un objeto con un único método `next()`, que nos devuelve un objeto literal con dos propiedades:

- `value`: contiene el elemento de la colección que estamos recuperando.
- `done`: es un booleano con valor `false` si todavía tenemos valores por recuperar, o `true` si ya hemos finalizado con todos los elementos.

El siguiente código muestra la estructura de un iterador. Como argumentos de entrada podemos tener lo que necesitemos, en este caso hemos considerado tener un contador indicando el inicio y el final. La función devuelve un objeto con el método `next()` implementado, el cual tiene acceso a las variables locales de la función por ser un closure.

El método `next()` devuelve los diferentes valores del contador sobre los que estamos iterando, asignando la propiedad done a `false` hasta que alcanzamos el límite, con lo que la propiedad `done` será `true`.

```
function myIterator(start, finish){
  // Iniciamos variables del closure
  let index = start;
  let count = 0;

  // Devolvemos objeto iterador con método next()
  return {
    next() {
      let result;
      // No hemos terminado de iterar
      if (index < finish) {
        // value le asignamos el índice, done es false porque no hemos
termiando.
        result = {value: index, done: false};
        index++;
        count++;
      } else { // Condición de salida
        // done es true, value le asignamos la cuenta.
        result = {value: count, done: true}
      }
      return result;
    }
  }
}
```

Pasamos a utilizar el iterador. Primero lo instanciamos, y luego ejecutamos el método `next()` hasta que la propiedad done sea `true`. En el siguiente código vemos como recuperamos los elementos de nuestro iterador de forma manual, para después instanciar un nuevo objeto y recorrerlo con un bucle.

```
let iterator1 = myIterator(2,5);
console.log(iterator1.next()); // {done: false, value: 2}
console.log(iterator1.next()); // {done: false, value: 3}
console.log(iterator1.next()); // {done: false, value: 4}
console.log(iterator1.next()); // {done: false, value: 3}

let iterator2 = myIterator(2,5);
let res = iterator2.next();
while (!res.done){
  console.log('Índice: ' + res.value);
  res = iterator2.next();
}
console.log('Número total: ' + res.value); // 3
```

Otro contexto donde podemos utilizar los iteradores es para ocultar referencias internas de un objeto, como por ejemplo un array u otro tipo de colección. Cuando hacemos pública la referencia a la colección, desde fuera del objeto podríamos cambiar el contenido sin seguir la lógica de negocio que implementa nuestra aplicación con el objeto.

El siguiente código implementa una clase *List* de forma rápida. Como campo privado tenemos un array para contener los elementos de la lista. No queremos que este array sea público, pero necesitamos recorrer los elementos de la lista.

```
/* Constructor de un objeto List para almacenar todo tipo de objetos */
function List(){
  if (!(this instanceof List))
    throw new Error("Debes invocar el constructor con el operador new");
  //Campo privado para los elementos de la lista.
  let _list = [];
  // Método para añadir un nuevo objeto en la lista
  this.add = function (obj){
    _list.push(obj);
    return _list.length;
  }
  // Método para borrar un objeto de la lista.
  this.remove = function (position){
    if (position >= _list.length || position < 0)
      throw new Error ("Intento de borrado fuera del límite.");
    return _list.splice(position,1);
  }
}
List.prototype = {};
List.prototype.constructor = List;
```

En una propiedad *iterator* vamos a crear el patrón. La propiedad debe permanecer en el cuerpo del constructor para que tenga acceso al campo privado. La propiedad es un *getter* en la que definimos un campo privado, *nextIndex*, con un contador inicializado a 0. Este campo privado se utiliza como índice al elemento que el iterador debe devolver. Este índice funciona como un closure.

El *getter* devuelve un objeto literal que nos permite recorrer los elementos del array, actuando como iterador sin publicar la referencia al array. Este objeto tiene acceso a *nextIndex*, por lo que sabe cuál es el siguiente elemento para entregar.

El método *next()* nos irá devolviendo los elementos del array. Si el índice es menor que el tamaño del array, el iterador devuelve el siguiente elemento de la lista. El índice es incrementado con el operador de post-incremento para que en la siguiente invocación se haya actualizado.

Si el índice es mayor que la longitud, no existe elemento que tengamos que devolver, por lo que solo devolvemos un objeto con propiedad *done* asignada a *true*.

```
Object.defineProperty(this, 'iterator', {
  get:function(){
    // Variable que mantiene la última posición recorrida por el array.
    let nextIndex = 0;
    // Devolvemos el objeto iterador para recorrer la lista.
    return {
      next: function(){ // Devuelve el siguiente objeto de la lista.
        return nextIndex < _list.length ?
            {value: _list[nextIndex++], done: false} :
            {done: true};
      }
    }
  }
});
```

Para iterar sobre *List* utilizando el iterador, creamos una lista con varios elementos. Necesitamos una variable con el iterador a partir de la propiedad, y sobre él recogemos el primer elemento de la lista con `next()`, aunque sabiendo que podría no existir por estar la lista vacía.

Para recorrer los elementos de la lista utilizamos un bucle `while` cuya condición está en función de la propiedad `done` de los objetos recuperados con el iterador.

```
const list = new List();
// Añadimos elementos a la lista
list.add({ property1: 'obj1', property2: 1 });
list.add({ property1: 'obj2', property2: 2 });
list.add({ property1: 'obj3', property2: 3 });

// Iteramos sobre la lista.
const { iterator } = list; // En un variable obtenemos un iterador sobre
la lista.
let obj = iterator.next(); // Obtenemos el primer elemento a través del
iterador si es que existe.
while (!obj.done) { // Mientras la propiedad "done" no es true debemos
iterar para obtener todos los objetos.
  const element = obj.value;
  // Mostramos el valor del objeto obtenido del iterador a través de la
propiedad value.
  $$result.log(`Objeto: ${element.property1} ${element.property2}`);
  obj = iterator.next(); // Conseguimos el siguiente objeto del iterador.
}
```

8. Ejemplo de uso de patrones. Carrito de la compra

Vamos a crear una aplicación para implementar un carrito de la compra utilizando los patrones de diseño. Lo vamos a realizar mediante fases para representar mejor la evolución.

Comenzamos creando una estructura de directorios dentro del directorio *js*:

- *mitienda*: Directorio principal para albergar el código.
- *mitienda/entitities*: contendrá los ficheros con las entidades que vamos a crear.
- *mitienda/shoppingcart*: contendrá la aplicación del carrito de la compra.

8.1. Excepciones

En el directorio *mitienda* declararemos las excepciones base que vamos a utilizar en el fichero *exceptions.js*. Básicamente reutilizaremos las clases de el ejercicio de productos que utilizamos en el documento anterior sobre *objetos ES5*. Las excepciones que reutilizamos son:

- *BaseException*
- *InvalidAccessConstructorException*
- *EmptyValueException*
- *InvalidValueException*

Importamos el fichero en el *index.html*.

```
<script src="./js/mitienda/exceptions.js"></script>
```

8.2. Entidades

Reutilizamos las clases creadas en el ejercicio de productos del documento *objetos ES5* con la jerarquía de clases *Product* y las clases hijas *Laptop*, *Camera*, *Tablet* y *Smartphone*. Lo hacemos en el fichero *js/mitienda/entities/products.js*.

Importamos el fichero en el index.html.

```
<script src="./js/mitienda/entities/products.js"></script>
```

8.3. Factoría abstracta

Dentro de nuestra jerarquía de objetos, vamos a cambiar la clase *Product* para que no se pueda instanciar objetos, transformándola en una clase abstracta.

Primero vamos a crear una nueva excepción que permita identificar el intento de creación de un objeto abstracto en el fichero *exceptions.js*.

```
//Excepción personalizada para clases abstractas.
function AbstractClassException(className) {
  let instance = BaseException.call(this, "Error: The class " + className
+ " is abstract.");
  instance.name = "AbstractClassException";
  instance.className = className;
  return instance;
}
AbstractClassException.prototype = Object.create(BaseException.prototype,
{
  constructor: {
    value: AbstractClassException,
    enumerable: false,
    writable: false,
    configurable: false
  }
});
```

Creamos la expresión IIFE con el cerrojo que utilizaremos para impedir la creación de instancias.

```
(function(){
  let abstractCreateLock = true; //Definición del cerrojo.

})(); //Invocamos la función global.
```

Vamos a mover la función *Product* y las modificaciones del prototipo en el cuerpo de la función anónima. Además, debemos cambiar el inicio de *Product* para añadir el chequeo del cerrojo.

```
function Product(serial,brand,model,price,taxPercentage = Product.IVA){
  //La función se invoca con el operador new
  if (!(this instanceof Product))
    throw new InvalidAccessConstructorException();
  if(abstractCreateLock)
    throw new AbstractClassException("Product");
  abstractCreateLock = true; //Reactivamos el cerrojo.
```

Para cada una de las subclases, primero tenemos que integrarlas dentro del cuerpo de la función anónima, pero con una pequeña modificación en la que desactivemos el cerrojo justo antes de invocar al superconstructor.

```
abstractCreateLock = false; //Desactivamos el cerrojo.
Product.call(this,serial,brand,model,price,taxPercentage);
```

Por último, importamos las funciones en el objeto `window`.

```
window.Product = Product;
window.Laptop = Laptop;
window.Camera = Camera;
window.Smartphone = Smartphone;
window.Tablet = Tablet;
```

Las funciones de testeo creadas previamente deben poder ejecutarse. Las de *Laptop*, *Camera*, *Smartphone* y *Tablet*, siguen funcionando igual que estaban implementadas. La función de

testeo para *Product* no debe poder instanciar objetos, generando la excepción *AbstractClassException*.

8.4. Implementación de *ShoppingCart*

Vamos a implementar el objeto *ShoppingCart* con la funcionalidad del carrito de la compra. Añadimos el fichero *shoppingcart.js* con las funciones del objeto. Lo creamos dentro de la carpeta *mitienda/shoppingcart*. Lo importamos en el *index.html*.

```
<script src="./js/mitienda/shoppingcart/shoppingcart.js"></script>
```

8.4.1. Excepciones

Vamos a crear una serie de excepciones que utilizaremos en la implementación. Las excepciones son:

- *ShoppingCartException*: Excepción genérica del carrito de la compra.
- *ProductShoppingCartException*: Un método necesita un argumento *Product* para poder ejecutarse.
- *ProductNotExistException*: El producto al que se quiera acceder no existe.
- *PositionOutBoundsException*: Hemos sobrepasado el límite del número de elementos del carrito.

```
function ShoppingCartException() {
  const instance = BaseException.call(this, 'Error: Shopping Cart
Exception.');
  instance.name = 'ShoppingCartException';
  return instance;
}
ShoppingCartException.prototype = Object.create(BaseException.prototype,
{
  constructor: {
    value: ShoppingCartException,
    enumerable: false,
    writable: false,
    configurable: false,
  },
});
```

```
function ProductShoppingCartException() {
  const instance = ShoppingCartException.call(this, 'Error: The method
needs a Product parameter.');
  instance.name = 'ProductShoppingCartException';
  return instance;
}
ProductShoppingCartException.prototype =
Object.create(BaseException.prototype, {
  constructor: {
    value: ProductShoppingCartException,
    enumerable: false,
    writable: false,
    configurable: false,
  },
});

function ProductNotExistException(product) {
  const instance = ShoppingCartException.call(this, `Error: The product
doesn't exist in the cart. ${product.serial}`);
  instance.name = 'ProductNotExistException';
  return instance;
}
ProductNotExistException.prototype =
Object.create(BaseException.prototype, {
  constructor: {
    value: ProductNotExistException,
    enumerable: false,
    writable: false,
    configurable: false,
  },
});

function PositionOutBoundsException() {
  const instance = ShoppingCartException.call(this, 'Error: The position
is out of bounds.');
  instance.name = 'PositionOutBoundsException';
  return instance;
}
PositionOutBoundsException.prototype =
Object.create(BaseException.prototype, {
  constructor: {
    value: PositionOutBoundsException,
    enumerable: false,
    writable: false,
    configurable: false,
  },
});
```

8.4.2. Constructor *ShoppingCart()*

Creamos un constructor con dos campos privados, uno para los productos, y otro para las cantidades de esos productos.

```
function ShoppingCart() {
  // La función se invoca con el operador new
  if (!(this instanceof ShoppingCart)) { throw new
InvalidAccessConstructorException(); }

  // Definición de atributos privados del objeto
  const _products = []; // array con los productos del carrito
  const _quantities = []; // array con las cantidades de cada producto
del carrito.
}
ShoppingCart.prototype = {};
ShoppingCart.prototype.constructor = ShoppingCart;
```

8.4.3. Método *addProduct()*

Necesitamos un método para añadir un nuevo producto en el carrito o incrementar su cantidad si el producto ya existe. Para facilitar la invocación haremos que la cantidad sea un valor por defecto.

Como necesitamos saber si un producto ya está añadido al carrito, primero vamos a implementar una **función interna** que, dado un producto, nos diga la posición en el carrito que ocupa. Utilizaremos para ello el método `findIndex()` del objeto `Array`.

```
function getProductPosition(product) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  return _products.findIndex((x) => x.serial === product.serial);
}
```

El método *addProduct()* añade un nuevo producto al carrito, pero tiene en cuenta que el producto ya existe para añadirlo o incrementar su cantidad.

```
this.addProduct = function (product, quantity = 1) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  const position = getProductPosition(product);
  if (position === -1) {
    _products.push(product);
    _quantities.push(quantity);
  } else {
    _quantities[position] = _quantities[position] + quantity;
  }
}
```

Vamos a obtener el número de productos diferentes introducidos en el carrito.

```
this.getNumberProducts = function(){
  return _products.length;
}
```

Testeamos los dos métodos que hemos añadido. Creamos diferentes tipos de objetos con números de serie diferentes. Al menos dos objetos deben tener el mismo número de serie.

```
let p1 = new Laptop("111-111-111","HP","EliteBook", 1000);
p1.processor = "i7 8 núcleos";
p1.memory = "16GB";
p1.hd = "SDD";
p1.size = "1TB";
p1.system = "Windows 10";
let p2 = new Camera("111-111-112","Canon","PowerShot", 500);
p2.type = "Digital";
p2.resolution = 7;
p2.size = 7;
let p3 = new Smartphone("111-111-113","Sansung","Galaxy", 500);
p3.memory = "8GB";
p3.storage = "64GB";
p3.resolution = "1024x1024";
p3.size = 7;
p3.system = "Android";
let p4 = new Tablet("111-111-114","Sansung","Galaxy", 500);
p4.memory = "8GB";
p4.storage = "64GB";
p4.resolution = "1024x1024";
p4.size = 15;
p4.system = "Android";
let p5 = new Laptop("111-111-111","HP","EliteBook", 1000);
p5.processor = "i7 8 núcleos";
p5.memory = "16GB";
p5.hd = "SDD";
p5.size = "1TB";
p5.system = "Windows 10";

let sc = new ShoppingCart();
sc.addProduct(p1,2);
sc.addProduct(p2,3);
sc.addProduct(p3);
sc.addProduct(p4,2);
sc.addProduct(p5);
console.log(sc.getNumberProducts()); //4
```

8.4.4. Iterador sobre productos

No queremos que los arrays con los productos y las cantidades estén disponibles directamente fuera del objeto del carrito, por lo que habilitamos un iterador para mostrar la información de los campos privados.

La propiedad `value` va a devolver un objeto literal por cada producto en el carrito indicando el objeto del producto y la cantidad que tiene asociada.

```
Object.defineProperty(this, 'products', {
  get() {
    let nextIndex = 0;
    return {
      next() {
        let value;
        if (nextIndex < _products.length) {
          value = {
            value: { product: _products[nextIndex], quantity:
_quantities[nextIndex] },
            done: false,
          };
          nextIndex += 1;
        } else {
          value = { done: true };
        }
        return value;
      },
    };
  },
});
```

8.4.5. Método *toString()*

Para implementar este método, el lugar de hacerlo directamente sobre los arrays, vamos a utilizar el iterador, de esta forma podemos heredar el método desde el prototipo. Vamos a añadir un parámetro *separator*, para indicar un tipo de separador entre cada producto.

```
ShoppingCart.prototype.toString = function (separator = '\n') {
  let str = '';
  const products = this.products;
  let product = products.next();
  while (!product.done) {
    str = `${str + product.value.product.toString()} Quantity:
${product.value.quantity}${separator}`;
    product = products.next();
  }
  return str;
};
```

Para testear el método, lo invocamos a continuación del código de testeo.

```
console.log(sc.toString());
```

8.4.6. Otros métodos heredados

Otros métodos que podremos crear serán para hacer los cálculos de totales de los carritos, además de obtener el precio sin impuestos y el total de impuestos. Utilizamos el iterador para realizar los cálculos.

```
ShoppingCart.prototype.getTotal = function () {
  let total = 0;
  const products = this.products;
  let product = products.next();
  while (!product.done) {
    total += product.value.product.price * product.value.quantity;
    product = products.next();
  }
  return total;
};
ShoppingCart.prototype.getTotalWithoutTaxes = function () {
  let total = 0;
  const products = this.products;
  let product = products.next();
  while (!product.done) {
    total += product.value.product.priceWithoutTaxes *
product.value.quantity;
    product = products.next();
  }
  return total;
};
ShoppingCart.prototype.getTaxes = function () {
  let total = 0;
  const products = this.products;
  let product = products.next();
  while (!product.done) {
    total += product.value.product.tax * product.value.quantity;
    product = products.next();
  }
  return total;
};
```

Para testearlo invocamos los tres métodos sobre el carrito ya creado.

```
console.log (sc.getTotal()); //6000
console.log (sc.getTotalWithoutTaxes()); //4740
console.log (sc.getTaxes()); //1260
```

8.4.7. Métodos auxiliares de comprobación productos

Es muy útil disponer de métodos que nos indiquen el estado del carrito en un determinado momento. El método *getQuantityProducts()* devuelve la cantidad de productos añadidos de un determinado producto del carrito. Si no existe generamos una excepción.

```
this.getQuantityProducts = function (product) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  const position = getProductPosition(product);
  if (position === -1) {
    throw new ProductNotExistException(product);
  }
  return _quantities[position];
};
```

El método *getQuantityProductPosition()* realiza la misma acción pero por posición.

```
this.getQuantityProductPosition = function (position) {
  if (position === 'undefined') throw new
EmptyValueException('position');
  if (position >= _products.length || position < 0) throw new
PositionOutBoundsException();
  return _quantities[position];
};
```

Por último, dada una posición obtenemos el producto de dicha posición.

```
this.getProduct = function (position) {
  if (position === 'undefined') throw new
EmptyValueException('position');
  if (position >= _products.length || position < 0) throw new
PositionOutBoundsException();
  return _products[position];
};
```

Testeamos los métodos creados.

```
console.log (sc.getQuantityProducts(p3)); //1
console.log (sc.getQuantityProductPosition(2)); //1
console.log (sc.getProduct(2).serial); //111-111-113
```

8.4.8. Modificación de las cantidades del carrito

Queremos poder modificar las cantidades de un producto en el carrito. El método *updateProduct()* recibe como argumento un producto y el nuevo número que tendrá de cantidad. Este método estará en el cuerpo de la función constructora al necesitar acceder a los arrays.

```
this.updateProduct = function (product, quantity = 1) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  if (quantity < 0) throw new InvalidValueException('quantity',
quantity);
  const position = getProductPosition(product);
  if (position !== -1) {
    _quantities[position] = quantity;
    if (_quantities[position] === 0) {
      _products.splice(position, 1);
      _quantities.splice(position, 1);
    }
  } else {
    throw new ProductNotExistException(product);
  }
};
```

Además de trabajar por producto también trabajamos por posición.

```
this.updateProductPosition = function(position, quantity = 1){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= _products.length || position < 0) throw new PositionOut
BoundsException();
  if (quantity < 0) throw new InvalidValueException("quantity",quantity);
  _quantities[position] = quantity;
  if (_quantities[position] === 0){
    _products.splice(position, 1);
    _quantities.splice(position, 1);
  }
}
```

Testeamos los métodos.

```
sc.updateProduct(p4,5);
sc.updateProductPosition(1,5);
console.log (sc.toString());
```

8.4.9. Métodos de borrado por producto

Queremos dos métodos de borrado de productos de carrito. En primer lugar, queremos un método que, dado un producto, elimine en 1 su cantidad. Si solamente su cantidad es de uno, elimina el producto completamente del carrito. Esta funcionalidad queda implementada en *removeProduct()*.

```
this.removeProduct = function (product) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  const position = getProductPosition(product);
  if (position !== -1) {
    _quantities[position] = _quantities[position] - 1;
    if (_quantities[position] <= 0) {
      _products.splice(position, 1);
      _quantities.splice(position, 1);
    }
  } else {
    throw new ProductNotExistException(product);
  }
};
```

Como complemento al método anterior, vamos a implementar un método *removeAllProduct()* que elimine completamente un producto del carrito.

```
this.removeAllProduct = function (product) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  const position = getProductPosition(product);
  if (position !== -1) {
    _products.splice(position, 1);
    _quantities.splice(position, 1);
  } else {
    throw new ProductNotExistException(product);
  }
};
```

Como testeo probamos a eliminar varios productos. Al menos eliminamos un producto con cantidad de 1.

```
sc.removeProduct(p5);
sc.removeProduct(p2);
sc.removeProduct(p3);
sc.removeAllProduct(p4);
console.log (sc.getNumberProducts()); //2
console.log (sc.toString());
```

8.4.10. Métodos de borrado por posición

Como complemento al borrado por producto, creamos un método de borrado por posición que recibe como argumento la posición en el carrito del producto que queremos eliminar para decrementar en uno dicha cantidad. Nuevamente si la cantidad actual es de 1, el producto es eliminado completamente del carrito.

```
this.removeProductPosition = function(position){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= _products.length || position < 0) throw new PositionOut
BoundsException();
  _quantities[position] = _quantities[position] - 1;
  if (_quantities[position] <= 0){
    _products.splice(position, 1);
    _quantities.splice(position, 1);
  }
}
```

La segunda versión de borrado por posición permite borrar completamente un producto del carrito en función de su posición.

```
this.removeAllProductPosition = function(position){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= _products.length || position < 0) throw new PositionOut
BoundsException();
  _products.splice(position, 1);
  _quantities.splice(position, 1);
}
```

Para testearlo tenemos el siguiente código.

```
sc.removeProductPosition(0,2);
sc.removeAllProductPosition(1);
console.log (sc.getNumberProducts()); //1
console.log (sc.toString());
```

8.4.11. Vaciado del carrito

Por último, necesitamos poder vaciar completamente el carrito.

```
this.clear = function(){
  _products.length = 0;
  _quantities.length = 0;
}
```

Lo testeamos invocando el método.

```
sc.clear();
console.log (sc.getNumberProducts()); //0
console.log (sc.toString());
```

8.5. Implementación del patrón Singleton

Para terminar, vamos a implementar el patrón Singleton con el objeto *ShoppingCart* para que en cada página solamente podamos tener una única instancia del objeto.

8.5.1. Estructura del patrón

Creamos la estructura del patrón en base la expresión IIFE. De momento devolvemos solamente un objeto vacío en la función *init()*.

```
const ShoppingCart = (function () {
  let instantiated;

  function init() {
    return {}
  }
  return {
    getInstance: function () {
      if (!instantiated) {
        instantiated = init();
      }
      return instantiated;
    }
  };
})();
```

8.5.2. Añadimos el constructor *ShoppingCart*

En el **cuerpo de la función anónima** debemos añadir el constructor junto con los métodos que ya hemos diseñado del prototipo.

Por último, debemos de crear una instancia con el operador `new`, y que *init()* devuelva dicha instancia. Vamos a congelar la instancia para que no sea modificable desde el exterior.

```
function init() {
  const sc = new ShoppingCart();
  Object.freeze(sc);
  return sc;
}
```

8.5.3. Testeo de Singleton

Utilizamos el mismo código de testeo, pero debemos crear la instancia utilizando el patrón.

8.6. Encadenamiento de métodos

Una herramienta muy útil que utilizan muchos frameworks en JavaScript es el encadenamiento de métodos. Esta técnica nos permite en una misma sentencia invocar diversos métodos sobre el mismo objeto, para ello, debemos devolver en cada método la instancia del propio objeto con `this`. En el siguiente código hemos modificado *removeProduct()* para que se pueda reutilizar varias veces.

```
this.removeProduct = function (product) {
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException();
  }
  const position = getProductPosition(product);
  if (position !== -1) {
    _quantities[position] = _quantities[position] - 1;
    if (_quantities[position] <= 0) {
      _products.splice(position, 1);
      _quantities.splice(position, 1);
    }
  } else {
    throw new ProductNotExistException(product);
  }

  return this;
};
```

Para testear la técnica invocamos al método varias veces en la misma sentencia. Al devolver la instancia del objeto, podemos reutilizarla en la siguiente invocación.

```
sc.removeProduct(p5).removeProduct(p2).removeProduct(p3);
```

Podemos utilizar esta técnica en cada uno de los métodos que no devuelvan ningún valor. La lista son *addProduct()*, *updateProduct()*, *updateProductPosition()*, *removeAllProduct()*, *removeProductPosition()* y *removeAllProductPosition()*.

Podemos probar a invocar todos estos métodos de forma encadenada.

9. Conclusión

En este apartado hemos visto cómo podemos estructurar el código en ES5 a través de patrones de diseño, con lo que facilitamos el mantenimiento y la reutilización del código, pero a costa de perder legibilidad del código. Ahora el código es un más difícil de interpretar que en las primeras versiones.

Todas las versiones de JavaScript utilizan estructuras de objetos basadas en prototipos. En la versión ES6 tenemos una sintaxis más amigable para trabajar con objetos, pero en esencia, la estructura de objetos sigue basándose en la propiedad *prototype* de las clases y la propiedad *__proto__* de las instancias que creemos.