Herencia (Parte II)

Polimorfismo

Competencias

- Aplicar el mecanismo de polimorfismo para la reutilización de componentes en el contexto de la Programación Orientada a Objetos.
- Utilizar closures para reducir el alcance de variables.

Introducción

Como observamos en lecturas anteriores, JavaScript es un lenguaje orientado a prototipos y no a clases, lo cual generaba algunos inconvenientes al momento de trabajar con la Programación Orientada a Objetos (POO), principalmente con el concepto de herencia, debido a lo amplio y complejo del código que se debe realizar para lograr aplicar los conceptos de POO, pero gracias a las nuevas actualizaciones creadas a partir de ES6 en la sintaxis de JavaScript, sobrellevar estos problemas y trabajar de forma adecuada hizo más interesante y agradable aprender POO con JavaScript. Por lo tanto, en esta lectura veremos conceptos claves para continuar trabajando con herencia, como el polimorfismo y los closures. El conocer estos conceptos y entenderlos, nos ayudará para poder construir software cada vez más robusto y con menor tendencia a errores.

Polimorfismo

Hasta el momento, trabajamos y logramos comprender el concepto de herencia y encapsulamiento, pero aún nos quedan conceptos por abordar en la Programación Orientada a Objetos (POO). Como es el caso de ***Polimorfismo***, en donde diferentes clases podrían definir el mismo método o propiedad, es decir, es la posibilidad de llamar métodos en común de diferentes objetos sin alterar los resultados. Por ejemplo, si estamos en un video juego, el cual será nuestra superclase, donde existen dos subclases como tanques y aviones, ambos objetos pueden tener propiedades y métodos con el mismo nombre, como se muestra a continuación:

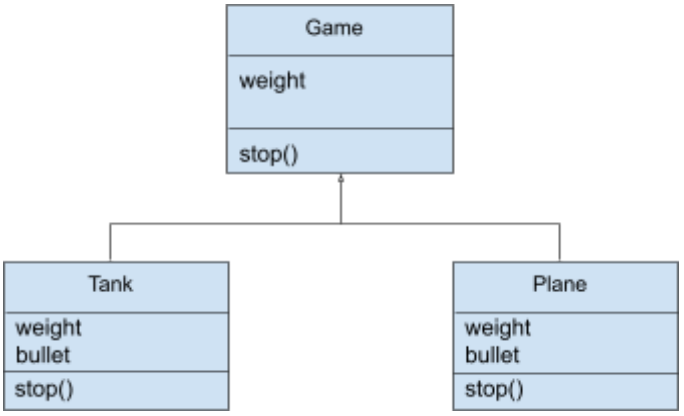

Imagen 1. Diagrama de clases UML.
Fuente: Desafío Latam

Como podemos observar en la imagen anterior, el método stop se encuentra disponible en las tres clases, tanto en la superclase o clase padre como en las subclases o clases hijas. La diferencia será que el método “stop” en la clase Game, deberá mostrar el mensaje: “El juego se encuentra detenido momentáneamente”, por ejemplo. Pero la propiedad deberá ser de ese objeto. Mientras que el método “stop” para la clase “Tank”, deberá indicar: “El Tanque se encuentra detenido...” y para la clase “Plane”, deberá decir: “El Avión apagó los motores”. Llevemos esto a código:

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un script.js. Seguidamente, en el mismo archivo debes definir la clase llamada "Game", la cual recibe como parámetro "weight" en su constructor y tendrá un método "stop" para retornar un mensaje.

```
class Game {
      constructor(weight){
         this.weight = weight;
      }
      stop(){
         return `El juego se encuentra detenido momentáneamente`;
      }
}
```

- ***Paso 2***: Definir las clases **"Tank" y "Plane"**, las cuales se extienden de **"Game",** es decir, heredan las características de la clase "Game" y reciben como parámetros **"weight"** (que heredan de Game), y otro parámetro distinto para cada clase como se muestra en el diagrama UML de clases. Seguidamente, se agrega el método "stop" a cada una de las clases, el cual, retornará un mensaje distinto para cada clase como se indica en el enunciado del ejemplo, quedando de esta forma nuestro código:

```
class Tank extends Game {
      constructor(weight,bullet){
         super (weight);
         this.bullet = bullet;
      }
      stop(){
         return `El Tanque se encuentra detenido...`;
      }
}
class Plane extends Game {
      constructor(weight,bullet){
         super (weight);
         this.bullet = bullet;
      }
      stop(){
         return `El Avión apagó los motores`;
      }
}
```

- ***Paso 3***: Ahora, después de realizar todas las clases, comenzaremos a generar las instancias de los objetos necesarios de esta forma:

```
let juego1 = new Game(20,300);
let tanque1 = new Tank(100,20);
let avion1 = new Plane(54,2000);
```

- ***Paso 4***: Llamar al método correspondiente, es decir, al método "stop", para ver qué sucederá con el mensaje que debe mostrar, es decir, si muestra un mensaje para instancia con su valor respectivo o simplemente muestra el mismo mensaje para todos.

```
console.log(juego1.stop());
console.log(tanque1.stop());
console.log(avion1.stop());
```

- ***Paso 5***: Al ejecutar el código realizado hasta el momento en la terminal con ayuda de Node, el resultado sería:

```
El juego se encuentra detenido momentáneamente
El Tanque se encuentra detenido...
El Avión apagó los motores
```

Se puede observar en el resultado mostrado anteriormente, como el mensaje cambia según cada instancia. Por consiguiente, cuando se hace la instancia de objeto de las clases Game, Tank y Plane, se puede ver que hemos sobrescrito el método stop en la superClase Game, esto es polimorfismo.

Ejercicio guiado: Polimorfismo

Implementar una aplicación que muestre películas y series bajo demanda, llamada "**MyApp**". Por consiguiente, partiremos con un modelo de herencia simple, como se muestra en la imagen número 2. En el modelo, en primer lugar tendremos una clase denominada **"Película"**, la cual, define el nombre de las películas, de esta clase heredarán las clases "**Largometrajes"** y **"Cortometrajes"**, cada una de estas tienen como atributo los tiempos de duración de cada una ("runtime"), siendo un método que permite obtener o modificar el valor, mediante "get" y "set".

Además, tendremos una clase llamada **"MyApp"**, la cual será la clase donde se ejecuta un método "play" (reproducir la película), este método "play" recibirá como parámetro una instancia de la clase película, en otras palabras, puede recibir un Largometraje, Cortometraje o una película en sí. Por lo tanto, esta clase no tendrá constructor alguno y solo el método directo "play" que retorna una cadena de texto o string con el mensaje: la película "nombre de la película" se está reproduciendo...tiene una duración de "tiempo de la película".

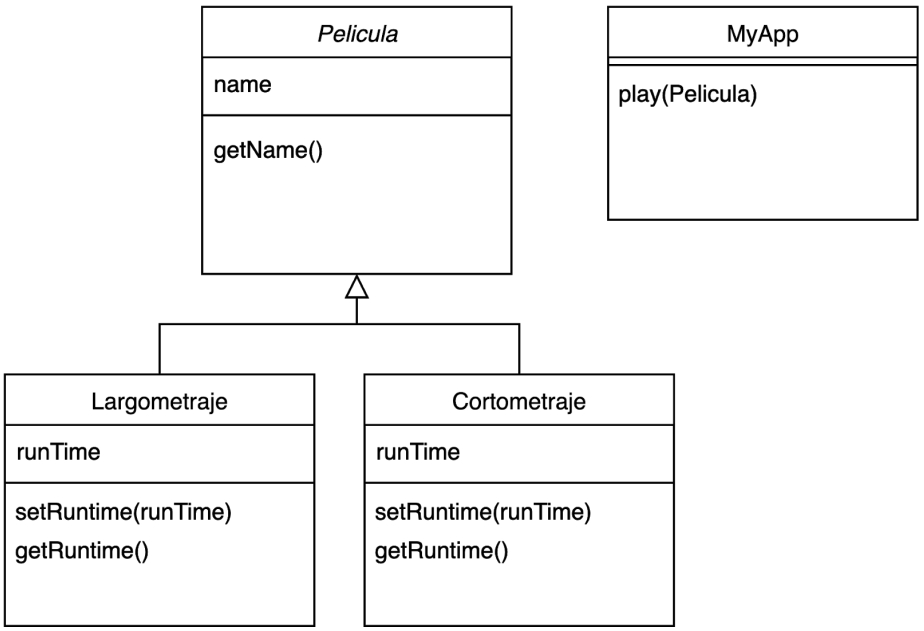

Imagen 2. Diagrama de clases.
Fuente: Desafío Latam

Para dar inicio a la solución del ejemplo planteado, se deben seguir los siguientes pasos:

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crear un archivo con el nombre de script.js. Luego, en este mismo archivo define la clase llamada "Película", la cual recibe como parámetro el título "title" en su constructor y tendrá un método "get" para regresar el título de la película.

```
class Pelicula {
   constructor(name){
       this._name = name;
   }
   getName() {
       return this._name;
   }
}
```

- ***Paso 2***: Definir la clase **"Largometraje"**, la cual extiende de **"Película",** es decir, hereda las características de la clase "Película" y recibe como parámetros "`name`" (el que hereda de Película), y **"runtime"** (la duración de la película). Seguidamente, se agregan los dos métodos indicados en la imagen número 2, un get para regresar el valor de la duración de la película y otro método set para modificar ese mismo valor, quedando de esta forma nuestro código:

```
class Largometraje extends Pelicula {
   constructor(name, runTime) {
       super(name);
       this._runTime = runTime;
   }
   getRuntime() {
       return this._runTime;
   }
   setRuntime(runTime) {
       this._runTime = runTime;
   }
}
```

- ***Paso 3***: Definir la clase **"Cortometraje"**, siendo hija de la clase "Película", recibiendo en su constructor como parámetros **name** y **runtime.** El parámetro **title** será enviado con la palabra clave "super" a la clase padre, mientras que el **runtime** se asignará a una variable en el constructor. Posteriormente, se deben agregar los dos métodos a la clase, como lo son el método `get runtime()` para regresar el valor de la duración de la película y el método `set runtime(runtime)` para modificar ese valor.

```
class Cortometraje extends Pelicula {
   constructor(name, runTime) {
       super(name);
       this._runTime = runTime;
   }
   getRuntime() {
       return this._runTime;
   }
   setRuntime(runTime) {
       this._runTime = runTime;
   }
}
```

- **Paso 4**: Definir la clase **"MyApp",** la cual define el método **"play"** recibiendo como parámetro un valor denominado "Pelicula" y retornando el mensaje: `` `la película ${Pelicula.getName()} se está reproduciendo...tiene una duración de ${Pelicula.getRuntime()}`; `` almacenado en una variable cualquiera, quedando nuestro código de esta forma.

```
class MyApp {
   play(Pelicula) {
       const result = `la película ${Pelicula.getName()} se está
reproduciendo...tiene una duración de ${Pelicula.getRuntime()}`;
       return result;
   }
}
```

- ***Paso 5***: Generar las instancias de los objetos necesarios, primero definiremos la instancia de Largometraje y Cortometraje, quedando, por ejemplo, de esta forma:

```
const largometraje = new Largometraje('Sin City', '105min');
const cortometraje = new Cortometraje('Hulk vs Wolverine', '20min');
```

- ***Paso 6***: Definir una instancia de la clase MyApp, así como una constante llamada playing, en la cual, llamaremos a la instancia de MyApp, llamando al método **play**, pasando como argumento el objeto **largometraje,** para luego imprimir la constante playing, quedando de esta forma.

```
const myApp1 = new MyApp();
const playing = myApp1.play(largometraje);
console.log(playing);
```

- ***Paso 7***: Al ejecutar el código realizado hasta el momento en el navegador web, el resultado en al consola seria:

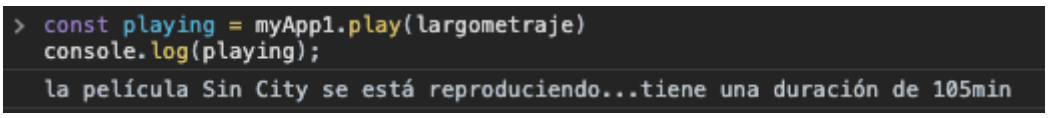

Imagen 3. Polimorfismo definición playing.
Fuente: Desafío Latam

- ***Paso 8***: Hacemos lo mismo que el paso anterior, pero para el objeto que instanciamos para Cortometraje, guardando en una variable el resultado del llamado y luego mostrando por consola la variable, quedando de esta forma.

```
const playing2 = myApp1.play(cortometraje);
console.log(playing2);
```

- ***Paso 9***: Al ejecutar el código realizado hasta el momento en el navegador web, el resultado en al consola seria:

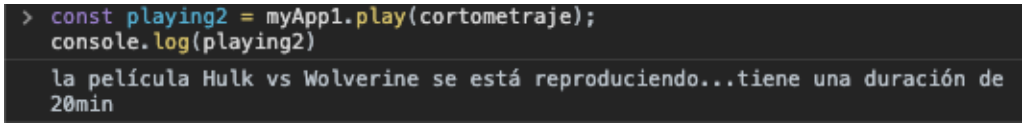

Imagen 4. Polimorfismo definición playing2.
Fuente: Desafío Latam

Con este ejemplo, hemos visto cómo podemos aplicar el concepto de Polimorfismo a nuestros desarrollos en JavaScript mediante clases y herencias de ES6. Ahora bien, si quieres observar otro ejercicio paso a paso, puedes ir al **Material Apoyo Lectura - Ejercicio de Polimorfismo,** ubicado en “Material Complementario”.

Ejercicio propuesto (1)

Desarrollar el HTML y posterior código en JavaScript, que permita a un usuario seleccionar de una lista dos posibles opciones en un menú desplegable, siendo las opciones cortometraje o largometraje. Por ende, el ejercicio consta de trabajar con el diagrama de clases anterior perteneciente a la imagen número 2; realizar el HTML y posteriores cambios al script.js, para que este sea capaz de generar el resultado mostrando en la imagen 5 y al momento de seleccionar un tipo de película, se debe mostrar el detalle de las películas definido en la instancia de cada objeto en el mismo documento web.

Imagen 5. Ejercicio.
Fuente: Desafío Latam

Recomendaciones:

- Implementa la etiqueta <select> con sus elementos <option> para desplegar las opciones.
- Crea un área para mostrar el resultado.
- Utiliza los addEventListener para escuchar los cambios mediante el evento “change” en la etiqueta del tipo <select>.
- Agrega un id a la etiqueta del tipo <select> y otro al área donde mostraras el resultado.

Closures

Antes de tocar el concepto de este capítulo, debemos recordar primeramente el proceso de las funciones, ya que siempre que se invoca una función se crea un nuevo ámbito para esa llamada. Por ende, la variable local declarada dentro de la función pertenece a ese ámbito y solo se puede acceder desde esa función. Por lo tanto, el alcance de la función se crea para una llamada de función, no para la función en sí, además, cada llamada a una función crea un nuevo alcance. Finalmente, cuando la función ha finalizado la ejecución, el alcance generalmente se destruye. Esto es importante recordarlo y entenderlo para poder aplicar y comprender los Closures.

En definitiva, un Closure es una función en donde no existen variables, es decir, el closure no tiene variables propias. En otras palabras, un closure es una especie diferente de objeto, ya que está compuesto por dos cosas, primero una función y luego del entorno donde se creó dicha función.

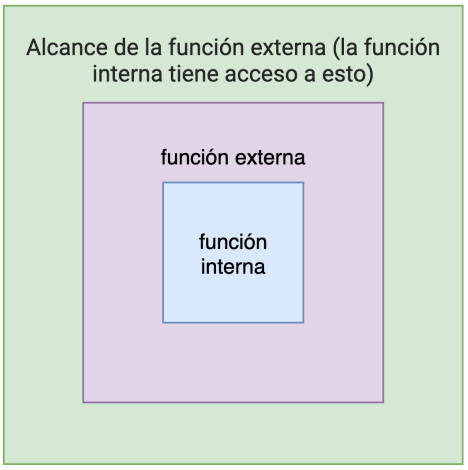

Imagen 6. Alcance de funciones.
Fuente: Desafío Latam

El esquema mostrado anteriormente nos ayuda a entender el concepto, es más o menos así, piensa que el alcance de la función interna está dado por el cuadrado interior (azul), esa función está contenida por otra función (morado), y el alcance de esta, viene dado por el color verde, que es donde ambas funciones deberían tener alcance. En conclusión, un Closure permite acceder al ámbito de una función exterior desde una función interior.
Un concepto que surge cuando empezamos a interiorizarnos en estos elementos, es el **scope,** lo cual no es más que el alcance que puede tener una variable en nuestro código, dicho de otra forma, el **scope** ("decide") a qué variables tienes acceso en cada parte de un código. Existen dos tipos de scope: global y local. Veamos un ejemplo para entender este concepto.

- ***Paso 1:*** En el primer caso, se declarará una función denominada global y dentro de ella simplemente mostraremos el valor de una variable declarada fuera de la función:

```
var a = 1;
function global() {
 console.log(a); // 1
}
```

- ***Paso 2:*** Como segundo paso se realizará el llamado a la función y posteriormente se volverá a mostrar el valor de la variable declarada al principio.

```
global();
console.log(a); // 1
```

Ejercicio guiado: Closures

- Utilizar dos funciones, una dentro de otra para observar el comportamiento de la variable dentro y fuera de ambas funciones.
- Utilizar variables pero dentro de la función y observar cómo reacciona el código.

Sigamos los siguientes pasos:

- ***Paso 1:*** inicializa una variable con el nombre de " a" y el valor número de "1", luego crea una función con el nombre de "**`global`**".

```
let a = 1;
function global(){}
```

- ***Paso 2:*** ahora dentro de la función global se llama a la variable global con el nombre de "a", seguidamente se crea una nueva función con el nombre de "**`interno`**", la cual, solamente mostrará la variable global "a" y se llamará dentro de la misma función global. Como se muestra en el código a continuación:

```
let a = 1;
function global() {
 console.log(a);
 function interno(){
   console.log(a);
 }
 interno();
}
```

- ***Paso 3:*** Al ejecutar el código anterior en la consola del navegador web, mediante las instrucción `global();` y luego volver a mostrar el valor de la variable local "a" con: `console.log(a);` encontraremos tres veces el número 1, es decir, la variable global siempre estará disponible no importa desde donde se invoque.

Ahora realicemos el ejercicio utilizando variables pero dentro de la función y observemos cómo reacciona nuestro código, para ello se deben realizar los siguientes pasos:

- ***Paso 1:*** crea una función con el nombre de "`local`", y dentro de ella inicializa una variable denomina "a" con el valor número de "2".

```
function local() {
 var a = 2;
}
```

- ***Paso 2:*** Ahora se debe mostrar dentro de la función local el valor de la variable creada bajo el nombre de "a" mediante un console.log(a). Seguidamente fuera de la función se debe hacer el llamado a la función local por su nombre y mostrar una vez más el valor de la variable "a" en la consola del navegador web.

```
function local() {
 var a = 2;
 console.log(a);
}
local();
console.log(a)
```

- ***Paso 3:*** Si el código anterior es ejecutado en al consola del navegador web, el resultado cambiará para ambas salidas, es decir, para el `console.log(a)` dentro de la función, el resultado será 2, pero para el `console.log(a)` invocado fuera de la función, el resultado será **`Uncaught ReferenceError: a is not defined`**. Esto se debe a que la variable "a" se encuentra declarada dentro de la función, es decir, de forma local, por lo tanto no se puede acceder a ella fuera de su entorno, sin importar si es declarada con let o con var.

- ***Paso 4:*** Posteriormente, se agrega una nueva función con el nombre de `interna` dentro de la función local y se intenta llamar a la variable creada en la función externa:

```
function local() {
 let a = 2;
 function interna () {
       console.log(a);
 }
 console.log(a);
 interna();
}
local();
```

- ***Paso 5:*** Al ejecutar el código anterior en la consola del navegador web, el resultado sería igual para ambas salidas, es decir, el número 2. Ya que la variable que se encuentra declarada dentro de la función con el nombre de local, ahora pasará a ser una variable global pero dentro del *scope* de esa función local, por esta razón la función interna puede llamar y/o utilizar la variable declarada en la función externa.

Sin darnos cuenta en el ejemplo anterior, aplicamos el concepto de ***Closure***, ya que se pudo acceder al ámbito de una función exterior desde una función interior. Pero hasta el momento trabajamos con los Closure implementado la sintaxis de ES5, vamos a ahora qué pasa con la sintaxis de ES6, para esto, definiremos una función que tendrá una variable dentro, y devolverá el valor de la misma, bastante simple, ¿no?, realicemos el código.

- ***Paso 1:*** Crear una función flecha en una constante con el nombre de "`local`" y dentro de ella inicializa una variable denomina "a" con el valor numérico de "2", mostrar esa misma variable mediante un `cosole.log(a)` dentro de la función local.

```
const local = () => {
 const a = 2;
 console.log(a);
}
```

- ***Paso 2:*** Ahora se debe hacer el llamado a la función local por su nombre y mostrar una vez más el valor de la variable "a" en la consola del navegador web en la parte externa o fuera de la función.

```
local();
console.log(a);
```

- ***Paso 3:*** Una vez ejecutamos nuestro código en la consola del navegador web, al llamar a la variable "a" dentro de la función local el resultado será "2", pero al llamar a la misma variable fuera de la función local, nos devolverá qué "a" no está definido, esto es porque "a" sólo existe dentro de la función, en consecuencia no podemos acceder a esta variable fuera de la función, en otras palabras, podríamos decir que estamos protegiendo la variable dándole una característica de privada. Como se muestra a continuación en el resultado:

```
2
Uncaught ReferenceError: a is not defined
```

Como se puede observar en el resultado anterior, al implementar la nomenclatura de ES6 seguimos obteniendo el mismo resultado que la de ES5. Para el scope global, definimos un nuevo código, el que nos permitirá acceder a elementos de forma global, entonces:

- ***Paso 1:*** inicializa una variable denominada "a" con el valor número 1. Luego crea una constante denominada "global" que contenga una función flecha y muestre el valor de la variable creada al inicio en la consola del navegador web.

```
const a = 1;
const global = () => {
 console.log(a);
}
```

- ***Paso 2:*** finalmente, realiza el llamado a la función "global" y muestra nuevamente en la consola del navegador web el valor que contenga de la variable "a"

```
global();
console.log(a);
```

Al ejecutar en nuestra consola web, y llamar a **"global"**, nos devolverá **1** al llamar a la función y además nos devolverá **1** al llamar a la variable, dado que no está dentro de la función y está disponible en cualquier parte de nuestro código. Es bastante simple comprender que significa el Scope, pero es necesario tenerlo claro, dado que es parte fundamental de lo que estamos viendo, Closures.

Ejercicio propuesto (2)

Traspasa el siguiente código que contiene un closure de ES5 a ES6

```
function crearFuncion() {
  var texto = "Variable dentro de funcion";
  function closure() {
      console.log(texto );
  }
  return closure;
}
var miFuncion = crearFuncion();
miFuncion();
```

Ejercicio propuesto (3)

Traspasa el siguiente código que contiene un closure de ES6 a ES5

```
const salto = () => {
  const height = 10;
  const interno = () => console.log(height);
  return interno();
}

var nuevoSalto = salto();
```

Ejercicio guiado: Closures y funciones con return

Generar una función que reciba dos números y dentro de ella se debe definir una función que realice la suma de ambos números. Para esto se deben seguir los siguientes pasos:

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella debes crear un archivo con el nombre de script.js. Seguidamente, en el archivo script.js, se debe crear una expresión de función con el nombre "`recibeNums`", almacenada en una constante, recibiendo como parámetros a num1 y num2, luego dentro de esta función, se debe retornar otra función pero esta vez denominada `suma`, la cual no recibe parámetros y retorna la suma de los parámetros recibidos en la primera función (num1 y num2).

```
const recibeNums = (num1, num2) => {
   return {
       suma: () => num1 + num2
   }
}
```

- ***Paso 2***: Ahora, al código anterior le agregaremos primeramente una definición de una constante llamada "result", la cual llama a ambas funciones y finalmente imprimimos el resultado usando el método "console.log()".

```
const result = recibeNums(2, 4).suma();
console.log(result);
```

- ***Paso 3***: Al ejecutar el archivo script.js con la ayuda de NodeJS, da como resultado la suma de ambos números, pero llamando a la función interna y esta tiene acceso a las variables, pese a que no se las pasamos como parámetro explícitamente (sum(x,y)).

```
6
```

Otra de las opciones que existen en los ***closures,*** es el paso de parámetros a nuestras funciones internas. Para comprender esto de una mejor manera, realicemos un nuevo ejemplo muy parecido al anterior, pero con nuevas modificaciones, por consiguiente crearemos un nuevo archivo y unas nuevas funciones.

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella debes crear un archivo con el nombre de script.js. Luego, en el archivo script.js, se debe crear una expresión de función con el nombre "`recibiendoNumeros`" almacenada en una constante, recibiendo como parámetros "a" y "b", luego dentro de esta función se deben retornar otras dos funciones, pero esta vez denominadas `restaNum` y `sumaNum` respectivamente, pero en esta ocasión, la función `restaNum` recibirá un parámetro denominado "c", retornando la resta del parámetro recibido menos el número uno (1), mientras que la otra función no recibirá parámetro alguno y retorna la suma de los parámetros recibidos en la primera función o principal (num1 y num2).

```
const recibiendoNumeros = (a, b) => {
   return {
       restaNum: (c) =>  c - 1,
       sumaNum: () => a + b
   }
}
```

- ***Paso 2***: Ahora, al código anterior le agregaremos primeramente una definición de una constante llamada "`restando`", la cual llamará a las funciones `recibiendoNumeros` y `restaNum,` y finalmente imprimimos el resultado usando el método "console.log()".

```
const restando = recibiendoNumeros(6, 8).restaNum(34);
console.log(restando);
```

- ***Paso 3***: Al ejecutar nuestro código en la terminal con la ayuda de NodeJS, veremos el resultado de la resta y que podemos pasar parámetros sin problemas.

```
6
```

No está limitado exclusivamente a funciones, dado que podemos también definir atributos dentro de las funciones y luego ser utilizados como cualquier otra variable. Continuemos con el ejercicio anterior y realicemos unos cambios.

- ***Paso 4***: Continuando desde el paso anterior, ahora modificaremos nuevamente nuestro código, agregando un atributo llamado "unNumero", el cual retornaremos y veremos su resultado, entonces nuestro código queda de esta forma.

```
const recibiendoNumeros = (a, b) => {
   const unNumero = 100;
   return {
       restaNum: (c) =>  c - 1,
       sumaNum: () => a + b,
       unNumero
   }
}
```

- ***Paso 5***: Al código anterior ya modificado, le agregaremos primeramente una definición de una constante llamada "`unNumeroNuevo`", la cual llamará a la función `recibiendoNumeros` y al atributo `unNumero`, finalmente imprimimos el resultado usando el método "console.log()".

```
const unNumeroNuevo = recibiendoNumeros(1, 2).unNumero;
console.log(unNumeroNuevo);
```

- ***Paso 6***: Al ejecutar nuestro código en la terminal con la ayuda de NodeJS, veremos el resultado:

```
100
```

De igual forma, si llamamos a la función sin argumentos, podemos acceder a los atributos, pero recuerda que esto es posible dado a que JavaScript no es un lenguaje altamente tipado, y no es recomendable, porque puede tener efectos inesperados. De igual forma, podemos ver el resultado si modificamos el console.log() como se muestra a continuación:

- ***Paso 7***: Al código anterior, ahora le modificaremos los valores en el argumento del llamado a la función `recibiendoNumeros`, quedando el código de la siguiente manera:

```
const unNumeroNuevo = recibiendoNumeros().unNumero;
console.log(unNumeroNuevo);
```

- ***Paso 8***: Al ejecutar nuevamente el archivo index.html en el navegador web, tendremos lo siguiente.

```
100
```

Esta es una buena forma de proteger los atributos de los objetos, dado que necesitamos acceder a la función externa para poder llegar a los datos que define la función interna.

Ejercicio propuesto (4)

Implementar una función en JavaScript que permita recibir un número, el cual puede ser incrementado o decrementado en uno de acuerdo al llamado de las funciones internas disponibles en la función principal y al valor de la instancia del objeto. Utiliza closures para dar respuesta al ejercicio.

Ejercicio guiado: Closures y clases de ES6

Continuando con nuestro proceso de aprendizaje, ahora vamos a trabajar con un ejercicio implementando Closures pero dentro del constructor de una clase en ES6. En este caso, nos entregan un código donde existen dos clases, una clase padre con el nombre de "Usuario" y una clase hija con el nombre de "Administrador", pero sin ningún tipo de protección sobre el atributo de la superClase, por lo tanto, se solicita implementar la técnica de Closures para proteger la propiedad del objeto padre convirtiéndola a privada.

```
class Usuario {
      constructor(nombre) {
         this.nombre = nombre;
      }
      saludar(){
         console.log("Bienvenido usuario: "+this.nombre);
      }
}

class Administrador extends Usuario {
      constructor(nombre){
         super(nombre);
      }
      saludar(){
         console.log("Panel de Administración - Usuario: "+
this.nombre);
      }
```

```
}

let adminUser = new Administrador ("Juan");
adminUser.saludar();
```

Para aplicar el método de Closure sobre el atributo de la clase “Usuario”, se deben seguir los siguientes pasos:

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un script.js.

- ***Paso 2***: Copia y pega el código o transcribirlo sobre el nuevo archivo creado en el paso anterior. Luego, lo primero que vamos a hacer es eliminar el “this” del atributo nombre y agregar la declaración como una variable con let, igualmente agregando el guión bajo a la propiedad “_”, para identificar que será un atributo privado y le asignamos el atributo que está llegando al constructor como parámetro. Por lo que el constructor quedaría primeramente de esta forma:

```
constructor(nombre) {
   let _nombre = nombre;
}
```

- ***Paso 3***: Ahora vamos a crear nuestras funciones internas que permitan retornar el valor de la variable privada que creamos anteriormente. Para ello, se creará primeramente una función “getNombre”, la cual retorna directamente el valor de la variable privada. Igualmente se agrega un método “setNombre” para modificar el valor de la variable privada. Es importante destacar que a estos métodos se les agregara la palabra reservada “this” para que sean accesibles desde el exterior.

```
 constructor(nombre) {
    let _nombre = nombre;
    this.getNombre = () => _nombre;
    this.setNombre = (nuevo_nombre) => _nombre = nuevo_nombre;
}
```

- ***Paso 4***: Ya con el paso anterior, se logra que no se acceda directamente a la propiedad del nombre y así no sufra modificaciones por error o mala asignación. Pero aún podemos seguir mejorando ese código agregando los métodos get y set a la propiedad nombre. Estos métodos apuntan directamente a las funciones que retornar o modifican el valor del atributo dentro del constructor.

```
get nombre(){
    return this.getNombre();
}
set nombre(nuevo_nombre){
    this.setNombre(nuevo_nombre);
}
```

- ***Paso 5***: Finalmente, ya podemos realizar las instancias necesarias y utilizar Node para mostrar el resultado directamente desde la terminal. Además, vamos a intentar acceder y modificar el atributo directamente después de crear la instancia para ver qué sucede.

```
let adminUser = new Administrador("Juan");
console.log(adminUser.nombre);
console.log(adminUser._nombre);
adminUser.saludar();
adminUser.nombre = 'Petra';
console.log(adminUser.nombre);
adminUser.saludar();
```

- ***Paso 6***: Al ejecutar todo el código anterior desde al terminal con ayuda del comando: `node script.js`, el resultado sería:

```
Juan
undefined
Panel de Administración - Usuario: Juan
Petra
Panel de Administración - Usuario: Petra
```

Se puede apreciar en el resultado mostrado anteriormente, como el resultado para la instrucción "`console.log(adminUser._nombre);`" es "`undefined`", ya que esa propiedad no tiene acceso al constructor de la clase porque es una variable local, perteneciente al constructor. Mientras que al intentar llamar a la propiedad por su nombre "`console.log(adminUser.nombre);`", automáticamente JavaScript buscará si existe un método get disponible para esa propiedad y como en esta caso existe, se ejecuta y realiza las operaciones que tenga asignadas, por ejemplo, llamar a la función que se encuentra dentro del constructor "`this.getNombre = () => _nombre;`" que si tiene acceso a la variable privada.

Ejercicio propuesto (5)

Partiendo del siguiente código, agregar Closures para proteger todos los atributos de la clase padre denominada "Producto", evitando así modificaciones externas.

```
class Producto {
   constructor(nombre, precio, sku){
      this.nombre = nombre;
      this.precio  = precio;
      this.sku  = sku;
   }
   generarCodigo(){
      console.log(this.nombre+this.precio+this.sku);
   }
}

class Movil extends Producto{
   constructor(nombre, precio, sku, sistemaOperativo) {
      super(nombre, precio, sku);
      this.sistemaOperativo = sistemaOperativo;
   }
   generarCodigo(){
    console.log(this.nombre+this.precio+this.sku+this.sistemaOperativo);
   }
}

let telef = new Movil('Samsung', 220000, 'GK234', 'Android');
```

Patrón de módulo

Competencias

- Reconocer el funcionamiento del patrón de módulo como contenedor para un contexto de ejecución.
- Aplicar el patrón de módulo en JavaScript para hacer el código más ordenado, mantenible y reusable.

Introducción

En el amplio espectro del desarrollo de software tenemos muchas formas de llegar a una solución de un problema, cuando la forma empieza a ser más frecuente y aceptada, muchas veces surge un patrón de desarrollo. Cabe mencionar que existen diversos patrones de diseño, los cuales se pueden aplicar dependiendo de la forma a la que se quiere llegar a la solución.

En este capítulo conoceremos y aplicaremos el patrón de módulo, para utilizar una forma de trabajo que nos ayudará a proteger nuestros datos y generar un software consistente con el paradigma orientado a objetos.

Funciones Autoejecutables

Como hemos visto hasta el momento, JavaScript implementa las funciones para codificar y dar soluciones a muchos problemas de la vida real mediante la programación, ya sean funciones anónimas, expresiones de funciones, funciones anidadas, funciones flechas, funciones constructoras, entre otras. Pero, ninguna de ellas tiene la posibilidad hasta el momento de ejecutarse por sí sola, para lograr esto JavaScript crea y deja a disposición de los programadores las expresiones de función ejecutadas inmediatamente (Immediately-invoked Function Expressions o por sus siglas: IIFE), las cuales, son funciones que se ejecutan tan pronto como se definen. Su sintaxis en ES5 es:

```
(function () {
      ...declaraciones...
})();
```

La sintaxis mostrada anteriormente, no es más que un patrón de diseño también conocido como función autoejecutable (Self-Executing Anonymous Function) y se caracteriza por estar conformada o compuesta en dos partes. La primera es la función anónima, mientras que la segunda parte crea la expresión de función cuya ejecución es inmediata (). Ahora bien, en ES6, se implementan las funciones flechas y se deja de utilizar la palabra reservada “function”, quedando:

```
(() => {
      ...declaraciones...
})();
```

Veamos unos ejemplos para comprender el funcionamiento de estas funciones autoejecutables, en este caso utilizaremos la sintaxis de ES5 y dentro de la función declararemos una variable, para mostrarla dentro y fuera de la IIFE, y ver cuál sería el resultado.

```
(function () {
   var interna = "Variable dentro de la IIFE";
   console.log(interna);
})();
console.log(interna);
```

Al ejecutar el código anterior en la consola del navegador web, el resultado que encontraremos será el siguiente:

```
Variable dentro de la IIFE
Uncaught ReferenceError: interna is not defined
```

Como se puede observar en el resultado mostrado anteriormente, al intentar mostrar la variable creada dentro de la función, el resultado será el esperado, es decir, el valor que tiene asignado, pero al intentar mostrar esa variable fuera de la función, el resultado indicará que esta variable no se encuentra definida. Si recordamos del tema anterior, esto se debe a que la variable declarada dentro de una función pasará a tener un scope o ámbito local y no podrá ser utilizada fuera de este ámbito local. Veamos ahora otro ejemplo más completo:

```
var lenguaje= "Ruby";
(function(){
  var lenguaje= "JavaScript";
  console.log(lenguaje + " es un lenguaje de programación");
})();
console.log(lenguaje);
```

En el código anterior, se puede indicar que una función IIFE crea un nuevo contexto donde podemos declarar variables a salvo de otras variables definidas en el ámbito global. Por ejemplo, si tenemos una variable global denominada "`lenguaje`", podemos definir otra variable con el mismo nombre dentro de la IIFE, sin miedo a que ambas sufren una colisión. Si ejecutamos el código anterior en la consola de nuestro navegador web, el resultado será:

```
JavaScript es un lenguaje de programación
Ruby
```

En el resultado mostrado anteriormente, se puede apreciar como la IIFE se ejecuta automáticamente después de ser definida y muestra por la consola dos mensajes. En este caso, dentro de la IIFE la variable "`lenguaje`" contiene el texto "JavaScript" , sin afectar a la variable `lenguaje` en su contexto global.

Ejercicio guiado: Funciones IIFE

Transformar el código de la sintaxis tradicional de ES5 a la nueva sintaxis de ES6 con funciones flechas para crear la función IIFE.

- ***Paso 1:*** Se debe crear la variable denominada lenguaje con un let y con el valor de "Ruby", para luego crear la estructura de la IIFE con la sintaxis de ES6.

```
let lenguaje= "Ruby";
(() => {})();
```

- ***Paso 2:*** Seguidamente se debe iniciar una variable con el nombre de lenguaje y el valor "JavaScript" pero dentro de la función IIFE. Seguidamente mediante un console.log, vamos a mostrar el siguiente mensaje: "`es un lenguaje de programación`" interpolando el valor de la variable creada dentro de la función IIFE.

```
let lenguaje= "Ruby";
(() => {
  lenguaje= "JavaScript";
  console.log(`${lenguaje} es un lenguaje de programación`);
})();
```

- ***Paso 3:*** Finalmente, fuera de la función IIFE se debe mostrar mediante un `console.log(lenguaje);` el valor que contenga la variable creada al inicio con el nombre de lenguaje. Obteniendo como resultado al ejecutar el código los mismos valores que en la versión realizada con la sintaxis de ES5:

```
JavaScript es un lenguaje de programación
Ruby
```

Ejercicio propuesto (6)

Para el siguiente código con una función del tipo IIFE, transforma la sintaxis de ES5 a ES6.

```
var resultado = (function (valor) {
      var name = "Jocelyn";
      return valor + " " + name;
})("Hola");
console.log(resultado);
```

Patrón de Módulo

El patrón módulo consiste en un módulo donde se encapsula toda la lógica de nuestra aplicación o proyecto. Dentro de este módulo estarán declaradas todas las variables o funciones privadas y sólo serán visibles dentro del mismo. Su principal atractivo es que resulta extremadamente útil para conseguir código reusable y sobre todo, "modularizar" nuestro código, para transformarlo en piezas reutilizables de código.

En Javascript el patrón de módulo es utilizado para emular el concepto de clases, de tal manera que podemos utilizar métodos públicos o privados como variables dentro de un objeto, protegiendo así partes de nuestro código de cambios innecesarios, utilizando closures y dejando solo una parte de la información disponible para consultar. Al exponer los datos de esta forma, tendremos una solución limpia para utilizar nuestro código en diversas partes de nuestra aplicación. Por lo tanto, este patrón utiliza funciones invocadas inmediatamente (IIFE, por su sigla en inglés) donde se retorna un objeto.

Ejercicio guiado: Contador con IIFE

Definir un módulo, el cual será una función invocada inmediatamente (IIFE) que dentro del cuerpo de la misma tenga una variable denominada “contador” y retorne dos funciones, una denominada “incrementaContador”, la cual tiene como responsabilidad incrementar el contador en 1 y otra llamada “reseteoContador”, que tiene como responsabilidad darnos el valor del contador, antes de devolverla a 0.

Para llevar esto al código, debemos implementar los siguientes pasos:

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella debes crear un archivo con el nombre de script.js. Seguidamente en el archivo script.js, inicializamos primeramente una constante denominada `moduloPrueba`, la cual, tendrá la IIFE como valor asignado, quedando el código de la siguiente manera.

```
const moduloPrueba = (() => {})();
```

- ***Paso 2***: Ahora dentro de la IIFE, lo primero es declarar la variable llamada “contador” e inicializarla en 0. Luego, retornamos dos funciones, una para incrementar el contador y otra para el reinicio del contador. Quedando el código:

```
const moduloPrueba = (() => {
   let contador = 0;
   return {
       incrementaContador: () => {},
       reseteoContador: () => {}
   };
})();
```

- ***Paso 3***: Queda por indicar los procesos dentro de las funciones `incrementaContador` y `reseteoContador`, en la primera retornamos el incremento de la variable llamada "contador" y en la segunda el mensaje: `` `valor del contador antes de reiniciar: ` ``, interpolando la variable contador y posteriormente llevándola a cero:

```
const moduloPrueba = (() => {
   let contador = 0;
   return {
       incrementaContador: () => {
           return contador++;
       },
       reseteoContador: () => {
         console.log(`valor del contador antes de reiniciar:
${contador}`);
           contador = 0;
       }
   };
})();
```

- ***Paso 4***: Al código anterior, le agregaremos primeramente el llamado a la variable que contiene la IIFE y luego la función que deseamos ejecutar, en este caso, primeramente ejecutaremos tres veces la función de incremento, luego una vez la función de decremento".

```
moduloPrueba.incrementaContador();
moduloPrueba.incrementaContador();
moduloPrueba.incrementaContador();
moduloPrueba.reseteoContador();
```

- ***Paso 5***: Al ejecutar el archivo script.js con ayuda de NodeJS, tendremos lo siguiente.

```
valor del contador antes de reiniciar: 3
```

Al trabajar de esta forma, otras partes que pudiéramos tener en el código, no podrán leer directamente el valor de "incrementaContador" o "reseteoContador". La variable "contador" está protegida del alcance global, por lo que se comporta como privada, dado que existe sólo dentro del módulo, la única forma de acceder a dicha variable es a través de las funciones.

Cuando trabajamos con este patrón es muy útil definir una forma de trabajo, el siguiente enfoque es muy utilizado, dado que tiene un nombre descriptivo (nameSpace) y posee variables “públicas” y “privadas”.

Ejercicio guiado: IIFE y el encapsulamiento

Definir las variables y métodos de manera “privada” para trabajar con ellos solo dentro de la función IIFE, luego exponer dentro del return de la IIFE elementos “públicos” que trabajarán en conjunto con los elementos privados, para acceder a ellos desde el exterior de la función IIFE.

- ***Paso 1***: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella debes crear un archivo con el nombre de script.js. Luego, en el archivo script.js, creamos primeramente una constante denominada , la cual tendrá la IIFE como valor asignado, quedando el código de la siguiente manera.

```
const miEspacio = (() => {})();
```

- ***Paso 2***: Ahora dentro de la IIFE, lo primero es declarar las variables llamadas “`miVariablePrivada` y `miMetodoPrivado`” e inicializarla en 0 a la variable privada. Posteriormente creamos una función denominada “`miMetodoPrivado`”, que recibirá un parámetro y a su vez mostrará mediante un console.log el valor de ese parámetro. Luego, dejamos el “return” de la función principal que retornara el objeto listo para utilizar. Quedando el código:

```
const miEspacio = (() => {
   let miVariablePrivada, miMetodoPrivado;

   // Una variable privada
   miVariablePrivada = 0;

   // Una función privada que muestra algo, al recibir un parámetro
   miMetodoPrivado = ( valor1 ) => {
       console.log( valor1 );
   };

   return {}
})();
```

- ***Paso 3***: Ahora dentro del return dejado en el código anterior, vamos a iniciar una propiedad llamada "`miVariablePublica`" con el valor "hola". Luego, crearemos una función denominada `miFuncionPublica` quien recibe como parámetro un valor cualquiera, por lo que indicamos el parámetro "valor2", para luego llamar a la función creada anteriormente con el nombre de `miMetodoPrivado`, quien mostrará por la consola del navegador web el valor del parámetro. Al mismo tiempo, dentro de esta función `miFuncionPublica`, incrementaremos en uno la variable creada anteriormente, denominada `miVariablePrivada` y la mostraremos directamente sobre esa fusión con ayuda de un console.log.

```
const miEspacio = (() => {
   let miVariablePrivada, miMetodoPrivado;
   miVariablePrivada = 0;
   miMetodoPrivado = ( valor1 ) => {
       console.log( valor1 );
   };

   return {
     // Una variable pública
     miVariablePublica: "hola",

     // Una función pública que utiliza los elementos privados
     miFuncionPublica: ( valor2 ) => {
       miVariablePrivada++;
       miMetodoPrivado( valor2 );
       console.log(miVariablePrivada);
     }
   }
})();
```

- ***Paso 4***: Al código anterior, le agregaremos primeramente el llamado a la variable que contiene la IIFE y luego la función que deseamos ejecutar, en este caso, ejecutaremos la función `miFuncionPublica` y le pasaremos un valor como parámetro.

```
miEspacio.miFuncionPublica("saludos");
```

- ***Paso 5***: Al ejecutar el archivo script.js con ayuda de NodeJS, vemos como el resultado de la ejecución de la misma ha cambiado el valor que habíamos definido en el método privado, retornando ahora el string que estamos definiendo más el valor actual de la variable privada que incrementó en uno.

```
saludos
1
```

Entonces, hemos visto porqué el patrón de módulo es una buena opción, es ideal cuando estamos comenzando con JavaScript, ya que nos da un concepto de verdadera encapsulación, en la forma que JavaScript nos provee para eso, es una de las principales ventajas. Por otra parte, también admite datos privados, que como ya vimos los elementos privados no pueden ser accesados desde fuera de la función.

Ejercicio propuesto (7)

Definir un archivo HTML, que tenga un selector de entradas (puede ser cine, recital, entre otras), pero al momento de seleccionar una de las entradas, esta se debe mostrar en la parte inferior del selector. Para esto, se debe implementar el patrón de módulo (IIFE) que permita obtener el resultado solicitado. Consiguiendo algo similar a la imagen mostrada a continuación.

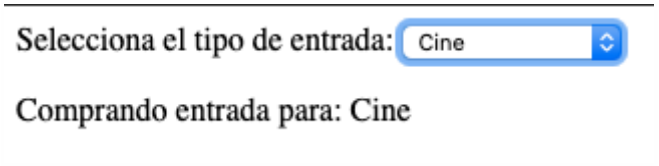

Imagen 7. Patrón de módulo - Ejercicio.

Recomendaciones:

- Implementa la etiqueta <select> con sus elementos <option> para desplegar las opciones.
- Crea un área para mostrar el resultado.
- Utiliza los addEventListener para escuchar los cambios mediante el evento “change” en la etiqueta del tipo <select>.
- Agrega un id a la etiqueta del tipo <select> y otro al área donde mostraras el resultado.

Resumen

A lo largo de esta lectura cubrimos:

- Aplicar el mecanismo de polimorfismo para la reutilización de componentes en el contexto de la Programación Orientada a Objetos.
- Utilizar closures para reducir el alcance de variables.
- Reconocer el funcionamiento del patrón de módulo como contenedor para un contexto de ejecución.
- Aplicar el patrón de módulo en JavaScript para hacer el código más ordenado, mantenible y reusable.

Solución de los ejercicios propuestos

1. Desarrollar el HTML y posterior código en JavaScript, que permita a un usuario seleccionar de una lista dos posibles opciones en un menú desplegable, siendo las opciones cortometraje o largometraje. Por ende, el ejercicio consta de trabajar con el diagrama de clases anterior perteneciente a la imagen número 2; realizar el HTML y posteriores cambios al script.js, para que este sea capaz de generar el resultado mostrando en la imagen 5 y al momento de seleccionar un tipo de película, se debe mostrar el detalle de las películas definido en la instancia de cada objeto en el mismo documento web.

Archivo index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width,
initial-scale=1.0">
      <title>Document</title>
</head>
<body>
      <div>
      Selecciona la película
      <select id="tipoPelicula">
            <option value="" selected>Selecciona una opción</option>
            <option value="cortometraje">Cortometraje</option>
            <option value="largometraje">Largometraje</option>
      </select>
      </div>
      <div>
      <p id="resultado"></p>
      </div>
      <script src="script.js"></script>
</body>
</html>
```

Archivo script.js

```
class Pelicula {
      constructor(title){
         this._title = title;
      }
      get title() {
         return this._title;
      }
}

class Largometraje extends Pelicula {
      constructor(title, runtime) {
         super(title);
         this._runtime = runtime;
      }
      get runtime() {
         return this._runtime;
      }
      set runtime(runtime) {
         this._runtime = runtime;
      }
}

class Cortometraje extends Pelicula {
      constructor(title, runtime) {
         super(title);
         this._runtime = runtime;
      }
      get runtime() {
         return this._runtime;
      }
      set runtime(runtime) {
         this._runtime = runtime;
      }
}

class MyApp {
      play(Pelicula) {
         const result = `la película ${Pelicula.title} se está
reproduciendo...tiene una duración de ${Pelicula.runtime}`;
         return result;
      }
}
```

```
const largometraje = new Largometraje('Sin City', '105min');
const cortometraje = new Cortometraje('Hulk vs Wolverine', '20min');
const myApp1 = new MyApp();
const playing = myApp1.play(largometraje);
const playing2 = myApp1.play(cortometraje);

tipoPelicula.addEventListener('change',function() {
      console.log(tipoPelicula.value);
      if (tipoPelicula.value == "cortometraje") {
         resultado.innerHTML = playing2;
      } else if(tipoPelicula.value == "largometraje"){
         resultado.innerHTML = playing;
      } else {
         resultado.innerHTML = "Debe seleccionar una de las dos
opciones"
      }
})
```

2. Traspasa el siguiente código que contiene un closure de ES5 a ES6

```
const crearFuncion = () => {
  let texto = "Variable dentro de funcion";
  const closure = () => {
      console.log(texto);
  }
  return closure;
}
let miFuncion = crearFuncion();
miFuncion();
```

3. Traspasa el siguiente código que contiene un closure de ES6 a ES5

```
function salto() {
  var height = 10;
  function interno () {
      return console.log(height)
  }
  return interno();
}

var nuevoSalto = salto();
```

4. Implementar una función en JavaScript que permita recibir un número, el cual puede ser incrementado o decrementado en uno de acuerdo al llamado de las funciones internas disponibles en la función principal y al valor de la instancia del objeto. Utiliza closures para dar respuesta al ejercicio.

```
const recibiendoNumero = (num) => {
      return {
         incrementar: () =>  ++num,
         decrementar: () => --num
      }
}

const resultado = recibiendoNumero(6).decrementar();
console.log(resultado);
```

5. Partiendo del siguiente código, agregar Closures para proteger todos los atributos de la clase padre denominada “Producto”, evitando así modificaciones externas.

```
class Producto {
      constructor(nombre, precio, sku){
      let _nombre = nombre;
      let _precio = precio;
      let _sku = sku;

      this.getNombre = () => {
            return _nombre;
      }
      this.setNombre = (nuevoNombre) => {
            _nombre = nuevoNombre;
      }

      this.getPrecio = () => {
            return _precio;
      }
      this.setPrecio = (nuevoPrecio) => {
            _precio = nuevoPrecio;
      }

      this.getSku = () => {
            return _sku;
      }
      this.setSku = (nuevoSku) => {
```

```
            _sku = nuevoSku;
      }
      }

      get nombre(){
      return this.getNombre();
      }
      set nombre(nuevoNombre){
      this.setNombre(nuevoNombre);
      }

      get precio(){
      return this.getPrecio();
      }
      set precio(nuevoPrecio){
      this.setPrecio(nuevoPrecio);
      }

      get sku(){
      return this.getSku();
      }
      set sku(nuevoSku){
      this.setSku(nuevoSku);
      }

      generarCodigo(){
      console.log(this.nombre+this.precio+this.sku);
      }
}

class Movil extends Producto{
      constructor(nombre, precio, sku, sistemaOperativo) {
      super(nombre, precio, sku);
      this.sistemaOperativo = sistemaOperativo;
      }
      generarCodigo(){
      console.log(this.nombre + this.precio + this.sku +
this.sistemaOperativo);
      }
}

let telef = new Movil('Samsung', 220000, 'GK234', 'Android');
```

6. Para el siguiente código con una función del tipo IIFE, transforma la sintaxis de ES5 a ES6.

```
let resultado = ((valor)=> {
      let name = "Jocelyn";
      return `${valor} ${name}`;
})("Hola");
console.log(resultado);
```

7. Definir un archivo HTML, que tenga un selector de entradas (puede ser cine, recital, entre otras), pero al momento de seleccionar una de las entradas, esta se debe mostrar en la parte inferior del selector. Para esto, se debe implementar el patrón de módulo (IIFE) que permita obtener el resultado solicitado. Consiguiendo algo similar a la imagen mostrada a continuación.

Archivo index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width,
initial-scale=1.0">
      <title>Document</title>
</head>
<body>
      <select id="salida">
         <option value="" selected>Seleccione una Opción</option>
         <option value="cine">Cine</option>
         <option value="recital">Recital</option>
         <option value="teatro">Teatro</option>
         <option value="concierto">Concierto</option>
      </select>
      <p>Comprando Entrada para: <span id="resultado"></span></p>
      <script src="script.js"></script>
</body>
</html>
```

Archivo script.js

```
const lugares = (()=>{
      return {
         salidaCine: () => {
            return "Cine";
         },
         salidaTeatro: () => {
            return "Teatro";
         },
         salidaRecital: () => {
            return "Recital";
         },
         salidaConcierto: () => {
            return "Concierto";
         },
         noSeleccionado: () => {
            return "Debe seleccionar una opción";
         },
      }
})();

salida.addEventListener('change',function () {
      switch (salida.value) {
            case "cine":
                  resultado.innerHTML = lugares.salidaCine();
            break;
            case "recital":
                  resultado.innerHTML = lugares.salidaRecital();
            break;
            case "teatro":
                  resultado.innerHTML = lugares.salidaTeatro();
            break;
            case "concierto":
                  resultado.innerHTML = lugares.salidaConcierto();
            break;
            default:
                  resultado.innerHTML = lugares.noSeleccionado();
            break;
      }
})
```