Programación

Introducción a la encapsulación

Cuarta semana

Octubre 2021

Cruz García, Iago

Introducción

La unidad anterior comenzamos a ver la encapsulación, como comentar código y las estructuras condicionales.

Esta semana veremos cómo trabajar con funciones, la encapsulación, de manera más detallada mediante el parametrizaje y la devolución de valores. Además, comenzaremos a ver el principio de la Programación Orientada a Objetos (POO) y las estructuras de repetición o bucles.

Recomendación: A partir de ahora es necesario que a la hora de realizar ejercicios nos acostumbremos a buscar información en la documentación oficial. En caso de no poder resolver las dudas con la documentación ofrecida, el siguiente paso será preguntar las dudas en el foro de clase.

Documentación oficial de JavaScript

Foro de la asignatura

Encapsulación avanzada

Como vimos, es mucho más cómodo trabajar si fragmentamos nuestro código en pequeñas acciones específicas. Para complementar esta manera de estructurar y trabajar nuestro programas, vamos a ver cómo podemos convertir realmente cada función en una especie de *caja negra* mediante el uso de parámetros:

```
function suma_numeros (numero1, numero2){
    var resultado = numero1 + numero2
    alert('El resultado es ' + resultado)
}

suma_numeros (3, 4)
```

Como veíamos la semana pasada, sabemos las partes que componen una función y, si recordamos, teníamos una parte en la cabecera, los '()' paréntesis, donde podíamos introducir parámetros. ¿Qué es un parámetro entonces?

Un parámetro, o argumentos, es un valor o un conjunto de valores que recibe el método cuando es invocado o llamado en otra instancia, permitiendo que una o más variables sean reutilizadas por diferentes funciones sin necesidad de reescribir código en cada método. Por ahora, trabajaremos con lo que parece algo muy simple, pero cuando veamos

polimorfismo entenderemos las ventajas que ofrece esta forma de estructurar nuestro código.

Otra de las piezas clave es trabajar con parámetros de salida. Esto comúnmente se le conoce como el valor que "devuelve" el método al ser invocado y terminar su función.

```
function suma_numeros (numero1, numero2){
   var resultado = numero1 + numero2
   return resultado
}

var total = suma_numeros (3, 4)
alert('Total: ' + total)
```

La palabra clave *return* indica que el método, una vez invocado y finalizado su función, "devolverá" o asignará el valor indicado en la posición donde fue "llamado". Es decir, en el ejemplo, al realizar la llamada de *suma_numeros(3, 4)* se le pasa por parámetros dos valores y este método devolverá la variable *resultado* para poder almacenarse en otra variable, tratándose como un valor más.

Programación Orientada a Objetos

Un objeto es una entidad que contiene datos (variables) y operaciones (funciones o métodos) para desarrollar un comportamiento concreto. Esta definición no se aleja mucho de los objetos reales: un coche es un objeto con una serie de atributos (número de ruedas, potencias, número de puertas…) y funciones (encendido del motor, acelerar, frenar…).

A la hora de programar, la ventaja de estas entidades es la capacidad de generar objetos con valores diferentes, almacenarlos y realizar operaciones con ellos, todo bajo el mantra de la **encapsulación**. Estos objetos los utilizamos con anterioridad, por ejemplo cuando usamos la función *console.log (texto)*, se compone del objeto **console** nativo de JavaScript y el método **log (argumentos)** que es una función del propio objeto,

Pero también podemos crear nuestros propios objetos. Veamos un ejemplo:

```
var Coche = {
    aceleracion: 30
    velocidad: 0
    Acelerar: function(){
        Velocidad = this.aceleración + this.velocidad // La referencia this.
                                                          indica que se usará el
                                                          atributo del objeto
    }
}
Coche.Acelerar()
console.log("velocidad= " + Coche.velocidad) //Mostrará "velocidad = 30"
```

De esta manera, creamos un objeto llamado **inline**, es decir, de una línea o un uso. La desventaja de esta forma es que no se puede utilizar múltiples veces. Veamos la manera en la que sí podemos crear más **instancias** de un mismo objeto.

```
var Pokemon = function(){
    this.nivel = 10;
    this.nombre = “”
    this.subir_de_nivel = function(){
        this.nivel++;
        console.log(nombre+” sube de nivel”)
    }
}

var pikachu = new Pokemon()
pikachu.nombre = “Pikachu”
pikachu.subir_de_nivel() //Mostrará por pantalla “Pikachu sube de nivel”

var mudkip = new Pokemon()
mudkip.nombre = “Mudkip”
mudkip.subir_de_nivel() //Mostrará por pantalla “Mudkip sube de nivel”
```

Este tipo de objetos ya pueden instanciarse múltiples veces y trabajar de manera paralela con cada uno. Para finalizar esta semana con **objetos**, vamos a ver como generar atributos privados y crear constructores. Los primeros permitirán crear variables que no son visibles desde fuera del objeto, evitando así modificaciones no deseadas. Lo segundo permitirá pasar por parámetro valores a las instancias de los objetos.

```
var Pokemon = function(nombre) {
        var nombre = nombre                    //El interprete sabe diferenciar
        var nivel = nivel                      //entre parámetro y variable

        this.subir_de_nivel = function() {
        nivel++
        console.log(nombre + " subió de nivel")
        }
}

var pikachu = new Pokemon("Pikachu")
var mudkip = new Pokemon()

mudkip.nombre = "Mudkip"                //No da error pero veremos las
                                        //consecuencias luego
pikachu.subir_de_nivel()
mudkip.subir_de_nivel()                 //Este aparecerá como Undefined
```

Si en vez de utilizar *this.nombre* se utiliza *var nombre* nuestra variable ya se considerará como privada y como vemos al final del ejemplo, aunque intentásemos acceder y cambiar el nombre al objeto *mudkip*, quedará como indefinido. Sin embargo, el objeto *Pikachu* es creado utilizando el parametrizaje del objeto para **construirlo**.

Estructuras de repetición

A la hora de crear un programa, muchos de los procesos son repetitivos: el menú de una aplicación o la carga de imágenes en un videojuego, por ejemplo. Para ello, utilizamos las estructuras de repetición que permiten ejecutar un bloque de código múltiples veces. En esta unidad vamos a ver las 3 estructuras básicas:

While()

La traducción de While del inglés, *mientras,* nos da una pista de que puede ocurrir cuando utilicemos esta construcción: el código encerrado entre '{}' llaves, se repetirá mientras la condición dentro de los '()' paréntesis sea verdadera, *true*.

```
var contador = 10
while (contador>0){
        console.log ("Quedan "+ contador + " vueltas")
        contador--
}
```

En este ejemplo, el bucle se realizará mientras la variable contador sea mayor que 0, por lo que serán 10 **iteraciones.**

Do{} While()

Una vez más, la propia traducción, *Hacer...mientras,* indica el resultado que se puede esperar. Esta vez, el bloque que se ejecutará estará entre el *Do* y el *While* encerrado entre '{}' llaves, mientras la condición entre '()' paréntesis sea **true**.

```
var contador = 10
do{
        console.log ("Quedan "+ contador + " vueltas")
        contador--
} while (contador>0)
```

En este ejemplo, el bucle se realizará mientras la variable contador sea mayor que 0, por lo que serán 10 **iteraciones,** igual que en el caso anterior.

La diferencia entre las dos estructuras es más básica que el número de iteraciones que puedan ofrecer: **while()** comprueba primero si la variable a comprobar cumple las condiciones, por lo que si la primera comprobación no es **true**, no se realiza ninguna iteración. Mientras tanto, **do{}while()** lo comprueba al final, por lo que la primera vez siempre se ejecuta el código.

```
var contador = 0 //Ahora contador empieza a 0
do {
        console.log("Quedan " + contador + " vueltas")
        contador--
} while (contador > 0) //Se ejecuta por lo menos una vez

while (contador > 0) { //No se ejecuta nunca
        console.log("Quedan " + contador + " vueltas")
        contador--
}
```

for ()

Esta estructura es más utilizada para realizar pasos o iteraciones un número de veces fijo, pues en la declaración de **for(),** es decir, entre paréntesis, habrá que establecer: la variable contador que se utilizará y podrá referenciar en el bloque, hasta que valor se repetirá el bloque y cómo se modificará el contador.

```
for (var i = 0; i<10;i++){
        console.log("Lleva "+ i + " vueltas")
}
```

Vamos a diseccionar la cabecera del **for()**:

- **var i:** se declara la variable. 'i' es una nomenclatura estándar, pero puede utilizarse cualquier nombre mientras siga las normas para variables.

- **i<10**: El bloque se repetirá hasta que 'i' sea igual o mayor a 10. Esta condición puede ser de cualquier tipo y puede incluir incluso otras variables.
- **i++:** Se incrementará la 'i' de uno en uno cada vez que se complete una iteración. Esto puede utilizar cualquier operación para asignar valores a la variable del contador

El bucle for es el más complejo de todos a la hora de establecer criterios y permite aunar en una misma línea el control sobre la condición de fin de bucle. ¿Pero qué pasaría si necesitamos romper estos bucles antes de que su condición se cumpla? Por un fallo o una comprobación concreta, por ejemplo. Tenemos dos maneras de hacerlo:

- La primera es modificar la variable directamente. En el caso de los ejemplos, cambiar 'contador' o 'i' a un valor que detiene el bucle es sencillo, evitaría las próximas iteraciones.
- Pero… ¿Y si no queremos que se ejecute más el bucle? Es decir, el código que queda hasta el final del bucle desde que cambiamos el valor se ejecutaría igual. Un ejemplo:

```
for (var i = 0; i<10;i++){
        i = 11
        console.log("Lleva "+ i + " vueltas") //Aunque cambie la i, se sigue
                                                //ejecutando
}
```

Sin embargo, contamos con una palabra clave, que ya vimos en estructuras condicionales, que nos permite saltarnos parte del código de un bloque encerrado entre '{}' llaves o un bucle: **break.**

```
for (var i = 0; i < 10; i++) {
        if (i == 5) {
        break //Al llegar aquí, la secuencia de ejecución saldría del bucle
        }
        console.log("Lleva " + i + " vueltas")
}
```

Anidamiento

Cabe destacar que todas las estructuras que hemos visto hasta el momento son anidables unas entre otras de forma infinita, es decir, podemos hacer bucles dentro de otros.

```
for (var i = 0; i < 10; i++) {
        for (var j = 0; j <10; j++){
          console.log(i + " x " + j +" = "+ (i*j))
        }
}
```

Conclusión

Vimos como encapsular nuestro código de manera mucho más eficiente y efectiva, pudiendo hacer prácticamente de todo en un método o función sin que nadie sepa que ocurre dentro pero realizando el trabajo que debería. Esto es clave a la hora de trabajar en equipo con otros desarrolladores para no perder tiempo en comprender todo el trabajo o modificaciones que se hagan al código.

Comenzamos la Programación Orientada a Objetos, con un principio muy básico de su uso y las posibilidades que brinda, además de esclarecer porqué funciones como *console.log()* funcionaban y cómo éramos capaces de usarlas.

Las estructuras de repetición nos van a permitir escribir un código mucho más rico y nos abre un abanico de posibilidades enormes.

Ahora que conocemos las 4 estructuras básicas (Entrada/Salida, Variables, Condicionales y Repeticiones) y cómo encapsular el código (métodos y objetos) nuestros programas pueden complicarse casi de manera infinita, pues estos 6 bloques tan sencillos, son la base para todo.

La próxima semana veremos cómo estructurar nuestros ficheros para que tenga sentido la encapsulación, otra manera de generar objetos de forma más eficiente y qué es la herencia y comenzaremos a interactuar con HTML.