JAVASCRIPT

CLASE 4

Material complementario

PROGRAMACIÓN CON FUNCIONES

CODER HOUSE

JavaScript: funciones

Funciones: definición

Las funciones son elementos que creamos con la intenciòn de agrupar instrucciones a ser utilizadas una o más veces, a lo largo del script. En la programación, el uso de funciones permite separar un problema en partes más pequeñas, independientes y reutilizables. Esto quiere decir que los programadores/as pueden programar las tareas de una solución por separado, para luego agruparlas con un orden determinado, y resolver así un problema más complejo.

JavaScript nos brinda un conjunto de funciones base que podemos emplear, algunas de estas ya las aprendimos: son ***prompt()***, ***console.log()*** y ***alert()***. Como sabemos, cada una de estas funciones realiza una acción específica: ***prompt*** solicita entradas, ***console.log*** realiza salidas por consola, y ***alert*** envía salidas mediante un cuadro de texto en el navegador. En ejercicios previos, combinamos estos elementos junto a estructuras condicionales y ciclos, para crear programas más complejos; dicho enfoque corresponde a la forma de trabajar con funciones.

Funciones: declaración

Para crear una función personalizada, tienes que seguir la siguiente forma:

```
function saludar() {
    console.log("¡Hola estudiantes!");
}
```

Distinguimos de la estructura previa los siguientes elementos

- Palabra reservada ***"function"***: término que utilizamos para indicar que estamos creando un función.

- Nombre de la función o identificador: en este caso, el nombre es ***"saludar"***; al igual que las variables, este nombre es definido por el programador/a e identifica a la función, determinando la denominación que usamos para emplear la misma en el script.
- Paréntesis ***()***: colocados luego del nombre de la función, nos permiten recibir valores a emplear dentro de la función (también llamados parámetros). En el ejemplo anterior, no se recibe dato alguno, por ende no hay nada dentro de los paréntesis. En ejemplos posteriores veremos cómo usar los llamados parámetros.
- Bloque ***{}***: al igual que con las estructuras, todas las instrucciones que definimos dentro del bloque serán interpretadas al momento de utilizar la función. Una vez definida la función, podemos emplearla en cualquier parte del programa; en la siguiente sección analizaremos cómo podemos hacerlo.

Funciones: llamada

Para utilizar funciones previamente definidas en el script, utilizamos su nombre y tipeamos los paréntesis; al hacer esto, en dicha instrucción se interpreta el bloque correspondiente a la función. A este empleo de la función, se lo conoce como ***llamada o invocación de la función***:

```
saludar();
```

Ventajas sobre el uso de funciones:

Las principales ventajas del uso de funciones en la programación son:

- Evitar instrucciones duplicadas (*Principio DRY*): no deberíamos tener dos instrucciones idénticas en el script; de ser ese el caso, debe transformarse dicha resolución en función
- Solucionar un problema complejo usando tareas sencillas (*Principio KISS*): si usamos funciones, dividimos un problema complejo en tareas independientes y reutilizables

- Focalizarse en tareas prioritarias para el programa (*Principio YAGNI*): nos enfocamos en desarrollar las funciones esenciales para el funcionamiento del programa. Luego, si es necesario agregar funcionalidades, sumamos otras funciones para agregar nuevas funcionalidades.
- Aporta ordenamiento y entendimiento al código: al tener el código fuente segmentado en funciones, es más sencillo leer todo el código (o lógica) que comprende el programa.
- Aporta facilidad y rapidez para hacer modificaciones: si es necesario cambiar, o sumar, o quitar instrucciones a una función, podemos hacerlo sin comprometer el funcionamiento del resto de las funciones, y de la totalidad del programa.

Caso de transformación a funciones:

Analicemos un caso donde estamos pidiendo tres veces consecutivas la misma entrada al usuario:

```
var nombreIngresado   = prompt("Ingresar nombre");
alert("El nombre ingresado es " + nombreIngresado);
var nombreIngresado   = prompt("Ingresar nombre");
alert("El nombre ingresado es " + nombreIngresado);
var nombreIngresado   = prompt("Ingresar nombre");
alert("El nombre ingresado es " + nombreIngresado);
```

Si tenemos esta situación, podemos transformar las dos instrucciones a función de la siguiente manera:

```
function solicitarNombre(){
    let nombreIngresado   = prompt("Ingresar nombre");
    alert("El nombre ingresado es " + nombreIngresado);
}
```

Ahora, si queremos utilizar la funciones tres veces, el código quedaría de la siguiente manera:

```
solicitarNombre();
solicitarNombre();
solicitarNombre();
```

Incluso podríamos utilizar una estructura de bucle (***for***, ***while*** o ***do...while***) para llamar a la función de forma consecutiva:

```
for (let i = 0; i < 3; i++) {
    solicitarNombre();
}
```

Parámetros

Cuando usamos ***console.log()*** o ***alert(),*** recordemos que escribimos entre paréntesis la variable o valor que queremos mostrar al usuario. A dicho elemento que se envía entre los paréntesis de la función se lo conoce como ***parámetro***.

Una función puede recibir ninguno, uno o más de un parámetro. Para especificar que es necesario enviar datos para emplear la función correctamente, detallamos en la declaración de la misma, entre los paréntesis, la cantidad de parámetros a "recibir", y el nombre con el que los identificamos dentro de la función separados por coma. Por ejemplo:

```
function conParametros(parametro1, parametro2) {
    console.log(parametro1 + " " + parametro2);
}
```

Luego, si queremos llamar a esta funciòn, el envío de datos se realiza de la siguiente manera:

```
conParametros("Hola", "Coder");
```

En importante identificar dos aspectos esenciales:

- El nombre del parámetro (por ejemplo: parámetro 1 y parámetro 2) determina cómo se utiliza este valor dentro de la función. El parámetro puede ser cualquier variable o un valor, pero dentro de la función lo usamos con el identificador correspondiente.
- En la llamada el orden de los parámetros importa: al llamar con parámetros, "Hola" es el primer valor enviado, y se reconoce como parámetro 1 dentro de la función; "Coder" es el parámetro 2. Si invertimos el orden de los valores enviados en la llamada (por ejemplo ("Coder", "Hola"), el identificador asociado a cada valor cambia; es decir que el elemento enviado en primer lugar siempre es parámetro 1, y el segundo elemento siempre es parámetro 2. Esta cuestión suele ser relevante al trabajar con parámetros de distinto tipo, u operaciones no conmutativas.

Caso de funciones con parámetros:

Las funciones que realizan cálculos matemáticos son un caso usual de función con parámetros. Analizemos ahora la declaracion de la función creada con el nombre ***suma:***

```
//Declaración de variable para guardar el resultado de la suma
let resultado = 0;
//Función que suma dos números y asigna a resultado
function sumar(primerNumero, segundoNumero) {
    resultado = primerNumero + segundoNumero;
}
```

Así, si queremos mostrar el resultado la instrucción comprende otra tarea, por lo cual puede ser conveniente realizar otra función para mantener las tareas lo más sencillas posibles (KISS):

```
//Función que muestra resultado por consola
function mostrar(mensaje) {
    console.log(mensaje);
```

```
}
```

Ahora que tenemos las funciones sumar y mostrar, podemos realizar las llamadas consecutivas; primero calculamos el resultado con sumar, y luego mostramos la salida por consola con mostrar:

```
//Llamamos primero a sumar y luego a mostrar
sumar(6, 3);
mostrar(resultado);
```

De esta manera, tenemos la división del problema "sumar y mostrar" en dos funciones, las cuales reciben parámetros con la intención de poder reutilizar este comportamiento para hacer diferentes cálculos y salidas al usuario

```
//Otras llamadas a sumar y mostrar
sumar(25, 5);
mostrar(resultado);
sumar(1, 2);
mostrar(resultado);
```

Resultado de la función

En el ejemplo anterior, usamos una variable resultado para asignar un nuevo valor a la misma cada vez que llamamos la función; pero si necesito obtener un valor al momento de llamar la función, en vez de utilizar un variable se recomienda emplear la palabra reservada ***return*** sobre el valor que quiero recuperar de la función, como por ejemplo:

```
function sumar(primerNumero, segundoNumero) {
    return primerNumero + segundoNumero;
}
let resultado = sumar(5, 8);
```

De esta manera, cada vez que llamo a la función obtengo el valor resultado del procesamiento de la función, sin necesitar una variable declarada previamente.

Hay que tener en cuenta dos cuestiones asociadas a ***return***:

1. Cuando se detecta un ***return*** en la función, se retorna el valor señalado y la función concluye, es decir que si existen instrucciones debajo del ***return*** no se interpretan.
2. Podemos codificar más de un ***return*** para establecer distintos valores de retorno de la función según distintos casos.

Para visualizar los aspectos mencionados tomemos el siguiente ejemplo como referencia, en el que la función calculadora retorna distintos resultados según el valor del parámetro operación:

```
function calculadora(primerNumero, segundoNumero, operacion) {
    switch (operacion) {
        case "+":
            return primerNumero + segundoNumero;
            break;
        case "-":
            return primerNumero - segundoNumero;
            break;
        case "*":
            return primerNumero * segundoNumero;
            break;
        case "/":
            return primerNumero / segundoNumero;
            break;
        default:
            return 0;
            break;
    }
}
console.log(calculadora(10, 5, "*"));
```

Ámbito de variable: scope

En la programación, según donde declaremos una variable, podemos determinar distintas restricciones de empleo, a las cuales se las conoce como ámbitos de la variable, contextos de la variable o scope.

Teniendo presente su lugar de creación, identificamos a las variables en dos tipos:

- Variables globales: si una variable se declara fuera de cualquier función o bloque, automáticamente se transforma en variable global, independientemente de si se define utilizando la palabra reservada var, o no. Por ejemplo:

```
let resultado = 0
function sumar(primerNumero, segundoNumero) {
    resultado = primerNumero + segundoNumero;
}
sumar(5,6);
//Se puede acceder a la variable resultado porque es global
console.log(resultado);
```

- Variables locales: cuando definimos una variable dentro de una función o bloque, la identificamos como una variable local, y existirá solamente durante la ejecución de esa sección. Si quisiéramos utilizarla por fuera, la variable no existirá para el intérprete. Por ejemplo:

```
function sumar(primerNumero, segundoNumero) {
    let resultado = primerNumero + segundoNumero;
}
//No se puede acceder a la variable resultado fuera del bloque
console.log(resultado);
```

El error que obtenemos al intentar acceder a la variable local resultado fuera del bloque (el cual podemos divisar desde la consola de desarrollo) es el siguiente:

En conclusión, usamos **variables globales** cuando queremos emplearlas en distintos bloques del programa, y **variables locales** cuando sólo son relevantes para la función o estructura actual.

Funciones anónimas y flechas

Las funciones son tan empleadas en la programación que JavaScript ofrece distintas herramientas para acotar su forma de escritura (definición). Estas formas distintas de escribir las funciones se conocen como ***funciones anónimas*** y ***funciones flechas***, y las desarrollaremos a continuación.

Función anonima

Una función anónima es aquella que se define sin nombre, y se utiliza con la finalidad de ser pasada por parámetro, o asignarla a variable. En el caso de referenciarla a una variable, puede llamarse usando el identificador de la variable declarada, es decir el nombre de la variable se emplea como nombre de la función, por ejemplo:

```
//Generalmente, las funciones anónimas se asignan a variables declaradas como constantes
const suma  = function (a, b) { return a + b };
const resta = function (a, b) { return a - b };
console.log(suma(15,20));
console.log(resta(15,5));
```

Pero ¿por qué necesito una función anónima si tengo las funciones normales? Como mencionamos, son más rápidas de escribir, y tienen una aplicación particular. Más

adelante en el curso, estudiaremos otras herramientas que nos permitirán crear interfaces HTML, detectar acciones del usuario y animar elementos; la mayoría de estos recursos reciben funciones como parámetro, y siendo esta función de uso exclusivo para el elemento cuestión se opta por declararla como anónima.

También cabe aclarar que si una función es compleja, es decir, cuenta con bastantes instrucciones, no se recomienda declararla como anónima ya que perderíamos el nombre, y si bien tenemos el identificado de la variable para reconocerla, la forma tradicional es más legible y conveniente en este caso.

Función flecha

Identificamos a las funciones flechas como funciones anónimas de sintaxis simplificada. Están disponibles desde la versión ES6 de JavaScript; no usan la palabra function, pero usamos => (flecha) entre los parámetros y el bloque:

```
const suma  = (a, b) => { return a + b };
//Si es una función de una sola línea con retorno podemos evitar escribir el cuerpo.
const resta = (a, b) =>  a - b ;
console.log(suma(15,20));
console.log(resta(20,5));
```

Pero ¿por qué necesito una función fecha si tengo las funciones normales y anónimas? Mencionamos que las funciones flecha son una forma más acotada de declarar funciones anónimas, incluso para funciones de una sola línea con un return podemos optar por codificar la función flecha sin cuerpo (es decir, omitiendo las llaves y el return porque se asumen por el intérprete). Esta forma más acotada permite que las funciones que reciban otras funciones como parámetro, compuestas de una sola línea, sean más legibles. Es común tener funciones de una sola línea para hacer un cálculo (suma, resta, etcétera) o realizar comparaciones reiterativas (igualdad de dos valores, desigualdad, entre otras).

Otra razón de aplicación de funciones flecha es emplear programación funcional en la construcción de aplicaciones modernas, Veamos entonces un ejemplo de cómo un

procesamiento del cálculo de precio se codifica empleando funciones flechas:

```
const suma  = (a,b) => a + b;
const resta = (a,b) => a - b;
//Si una función es una sola línea con retorno y un parámetro puede evitar escribir los ()
const iva   = x => x * 0.21;
let precioProducto  = 500;
let precioDescuento = 50;
//Calculo el precioProducto + IVA - precioDescueto
let nuevoPrecio = resta(suma(precioProducto, iva(precioProducto)), precioDescuento);
console.log(nuevoPrecio);
```

Como podemos notar, para calcular nuevoPrecio empleamos 3 funciones:

1. La función resta nos permite restar el monto de descuento al valor del primer parámetro.
2. Pero el valor de la resta se obtiene desde la llamada suma, es decir, restamos al valor que obtenemos llamando a suma como primer parámetro.
3. Luego suma el primer parámetro que es el precio del producto, y el valor a sumar es el resultado del cálculo de iva mediante la función iva.

Esta operación realiza el siguiente cálculo matemático gracias a estas llamadas:

((500 + (500 * 0.21)) - 50)