<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

FULL STACK PYTHON
Clase 15

Javascript 3

BA

Programación modular con funciones

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 14

Condicionales y Ciclos

- Control de flujos.
- Condicional. ¿Qué es?
- Operadores lógicos y de comparación. Uso en los condicionales?
- Ciclos. ¿Qué son? Tipos y diferencias entre sí.
- Cómo combinar operadores lógicos y ciclos.

Clase 15

Programación modular con funciones

- Funciones. ¿Qué son? Scope global y local.
- Programación modular vs. Funciones.
- Función anónima y función flecha.
- Callbacks y clausuras.

Clase 16

Objetos

- Objetos. ¿Qué son y cómo se usan?
- Propiedades y métodos.
- Función constructora.
- El objeto String y sus métodos.
- El objeto Math, sus propiedades y métodos.

Funciones

Las **funciones** son estructuras esenciales dentro del código. Una función es un grupo de instrucciones que constituyen una **unidad lógica** del programa y resuelven un problema muy concreto. Presentan varias ventajas, entre ellas las de permitir dividir un problema complejo en partes menores y más simples, reutilizar código en el mismo o en otro programa, simplificar la depuración, etcétera.
JavaScript proporciona al usuario una serie de funciones implementadas y listas para utilizar. Sin embargo, no es difícil encontrar situaciones en las que necesitamos realizar alguna tarea para la cual no existe una función disponible, y debemos utilizar los mecanismos que nos proporciona JS para construir nuestras propias funciones.

Programación modular

La **metodología de división** por módulos se conoce habitualmente como “divide y vencerás” y en programación se llama **Desarrollo Top Down**.

¿Cuál será la **estrategia** para resolver problemas? Pensar en el problema general e ir **descomponiéndolo en sub-problemas** (sub-algoritmos). A su vez, estos subproblemas se podrán seguir dividiendo hasta llegar a un subproblema lo bastante simple como para poder resolverse de forma sencilla.

Abstracción

Podemos definir la **abstracción** como el aislamiento de un elemento de su contexto o del resto de los elementos que lo acompañan. En programación la abstracción está relacionada con “qué hace”.

Concretamente, la abstracción se produce cuando creamos módulos.

Lo **importante**, para entender el concepto de abstracción, es comprender que **cada módulo es independiente de los demás módulos** (**bajo acoplamiento**) y que es **ideal que realice una sola tarea** (**alta cohesión**).

Los módulos son independientes entre sí, aunque algunos pueden necesitar colaborar con otros, o trabajar de forma conjunta.

Funciones

Las funciones nos permiten agrupar líneas de código en tareas con un nombre (subprograma), para que posteriormente podamos referenciar ese nombre para realizar dicha tarea. Algunas razones para declarar funciones:

- **Simplificación:** Cuando un conjunto de instrucciones se va a usar muchas veces, se crea una función con esas instrucciones y se llama la cantidad de veces que sea necesario, reduciendo un programa complejo en unidades más simples.
- **División:** Una función me permite modularizar, es decir, armar módulos. De esta manera un equipo puede dividir el trabajo en partes. Cada integrante realiza una función, para luego integrarlas en un programa principal más grande.
- **Claridad:** Usando funciones un programa gana claridad, aunque esa función solo se llame una vez.
- **Reusabilidad:** Una función es reutilizable, sólo es necesario cambiar los valores de entrada.

Funciones

Para usar funciones es necesario hacer dos cosas:

- **Declarar la función**: crear la función es **darle un nombre**, definir los datos de entrada (opcional) e indicar las tareas (instrucciones) que realizará y qué valor retornará (opcional).
- **Ejecutar la función**: «Llamar» (invocar) a la función para que realice las tareas del código que aloja. Se puede invocar una misma función la cantidad de veces que se necesita desde el programa principal.

```
// Declaración de la función "saludar"
function saludar() {
 // Contenido de la función
  console.log("Hola, soy una función")
}
```

Primer paso:
Declarar la función

```
// Ejecución de la función
saludar()
```

Segundo paso:
Ejecutarla

Funciones

El **nombre** de la función tiene que ser significativo y describir lo que hace. Los nombres de las funciones tienen las mismas características que los de las variables. Idealmente deben ser:

- Nombres simples, claros.
- Representativos de la tarea que realiza la función.
- Verbos en infinitivo (-ar, -er, -ir).
- Si es más de una palabra, utilizar la nomenclatura **camelCase**.

Es necesario definir los datos de entrada (si existen) e incluir las instrucciones necesarias para que realice su tarea. Opcionalmente se puede definir qué valor retornará.

Funciones | Ejemplo

Este código muestra la tabla de multiplicar por 5.

```
for (i = 1; i <= 10; i++) {
    console.log("1 x", i, "=", 5 * i)
}
```

Este código muestra la tabla de multiplicar por 5 tres veces. Funciona, pero usa demasiado código, repetido.

```
// Primera vez
for (i = 1; i <= 10; i++) {console.log("5 x", i, "=", 5 * i)}
// Segunda vez
for (i = 1; i <= 10; i++) {console.log("5 x", i, "=", 5 * i)}
// Tercera vez
for (i = 1; i <= 10; i++) {console.log("5 x", i, "=", 5 * i)}
```

Solución con bucle y función. La función *tablaDelCinco()* usa un **for** de 10 iteraciones. El otro **for** ejecuta la función 3 veces.

```
//Declaración de la función tablaDelCinco()
function tablaDelCinco(){
  for (i = 1; i <= 10; i++){console.log("5 x", i, "=", 5 * i)}
}
//Bucle que ejecuta 3 veces la función tablaDelCinco()
for (let i = 1; i <= 3; i++) {tablaDelCinco()}
```

Funciones | Clasificación

Según reciban o no datos, y devuelvan o no valores, las funciones se pueden clasificar en:

Funciones sin parámetros:

- Que no devuelven valores
- Que devuelven valores

Funciones con parámetros:

- Que no devuelven valores
- Que devuelven valores

Funciones | Parámetros y Argumentos

Los **parámetros** son las variables que ponemos cuando se define una función. En la siguiente función tenemos dos parámetros “a” y “b”:

```
function sumar(a, b){
    console.log(a + b)
}
```

Los **argumentos** son los valores que se pasan a la función cuando ésta es invocada, “7” y “4” en el ejemplo:

```
var suma = sumar(7, 4) //Pedimos valores
```

Dentro de la función, los **argumentos** se copian en los parámetros y son usados por ésta para realizar la tarea.

Funciones | Parámetros y Argumentos

Esta función tiene un sólo **parámetro** que indica hasta qué valor calculará:

```
// Declaración
function tablaMultiplicar(hasta) {
      for (var i = 1; i <= hasta; i++)
        console.log("1 x", i, "=", 1 * i)
}
```

```
//Ejecución
tablaMultiplicar(4)
```

En este ejemplo la función muestra un texto concatenado a un **argumento** pasado por **parámetro**:

```
// Declaración
function saludarDos(miNombre){
    console.log("Hola " + miNombre)
}
```

```
//Ejecución
saludarDos("Codo a Codo") //Argumento fijo
var nombre= prompt("Ingrese su nombre")
saludarDos(nombre) //Argumento variable
```

Funciones | Parámetros múltiples

Cuando se utilizan parámetros múltiples hay que respetar el orden en que los declaramos y el de los argumentos usados al llamarla. Esta función tiene dos parámetros: el valor de la tabla a generar y hasta qué valor calculará.

```
// Declaración
function tablaMultiplicar(tabla, hasta) {
      for (var i = 1; i <= hasta; i++)
        console.log(tabla + " x " + i + " = ", tabla * i)
}
```

```
// Ejecución
tablaMultiplicar(1, 10) // Tabla del 1, calcula desde el 1 hasta el 10
tablaMultiplicar(5, 8) // Tabla del 5, calcula desde el 1 hasta el 8
```

Funciones | Parámetros múltiples

Ejemplo con tres parámetros. Se evalúa la mayoría de edad de una persona:

```
// Declaración
function mayoriaEdad(miApellido, miNombre, miEdad){
    console.log("Apellido y nombre: " + miApellido + ", " + miNombre)
    if (miEdad >= 18) {
        console.log("Es mayor de edad " + "(" + miEdad + ")")}
        else{
        console.log("No es mayor de edad " + "(" + miEdad + ")")}
    }
```

```
//Ejecución
var ape= prompt("Ingrese su apellido")
var nom= prompt("Ingrese su nombre")
var edad= prompt("Ingrese su edad")
mayoriaEdad(ape, nom, edad)
```

*Esta función recibe tres parámetros y en función del valor de uno de ellos (**miEdad**) determina si la persona es mayor de edad (>=18)*

Parámetros predeterminados

Los parámetros predeterminados de función permiten que los parámetros con nombre se inicien con valores predeterminados si no se pasa ningún valor o undefined.

En JavaScript, los parámetros de función están predeterminados en undefined. Sin embargo, a menudo es útil establecer un valor predeterminado diferente.

```
function multiplicar(a, b = 1) {
    return a * b;
  }

console.log(multiplicar(5, 2));  // salida: 10
console.log(multiplicar(5));  // salida: 5
```

Funciones | Devolución de valores

Una función puede devolver información, para ser utilizada o almacenada en una variable. Se utiliza la palabra clave **return**, que regresa un valor y finaliza la ejecución de la función. Si existe código después del **return**, nunca será ejecutado. Puede haber más de un **return** por función.

```
// Declaración
function sumar(a, b) {
  return a + b // Devolvemos la suma de a y b al exterior de la función
}
```

```
// Ejecución
var a = 5, b = 5
var resultado = sumar(a, b) // Se guarda 10 en la variable resultado
console.log("La suma entre "+ a +" y "+ b +" es: "+ resultado)
```

Funciones | Devolución de valores

Veamos dos funciones que hacen lo mismo, una retorna valores y otra no:

```
function sumar(num1, num2){
    var suma = num1 + num2
    console.log("La suma es " + suma)
}
sumar(2,5)
```

Esta función muestra “La suma es ...” en la consola, pero no retorna ningún valor al programa.

```
function sumarDos(num1, num2){
    var suma = num1 + num2
    return suma
}
n1 = 2
n2 = 3
var resultado = sumarDos(n1, n2)
console.log("El resultado es: " + resultado)
```

En este caso la función devuelve un valor, y se almacena en una variable llamada **resultado** que contiene la suma de dos valores realizada por la función **sumarDos**.

Funciones | Devolución de valores

Otra alternativa es hacer que la función guarde directamente el resultado que devuelve en una variable:

```
var suma = function sumarTres(numero1, numero2) {
    return numero1 + numero2
}
console.log(suma(40, 15))
```

Al retornar un valor, éste se guarda en la variable suma.

```
var numeroMaximo = function (valor1, valor2) {
    if (valor1 > valor2) {  return valor1 }
    return valor2
}
var v1 = parseInt(prompt("Ingrese un número entero"))
var v2 = parseInt(prompt("Ingrese otro número entero"))
console.log("El número máximo es:", numeroMaximo(v1,v2))
```

En este caso se piden dos valores y si la condición no se cumple se asume que el **valor2** es el máximo (no es necesario un **else**)

Funciones | Función flecha (arrow Function)

En **JavaScript** existe la forma resumida de escribir las funciones. Se llaman **funciones flecha**, en alusión a **=>**. Permiten definir funciones de manera más fácil, breve y rápida, aunque están limitadas a funciones más simples. Para crear estas funciones flecha partiremos del ejemplo:

```
// Función tradicional
function cuadrado(x){
    return x*x
}
console.log(cuadrado(2))
```

```
// Función Flecha (Arrow)
var aCuadrado = x => x*x
console.log(aCuadrado(2))
```

x es el parámetro. A la derecha de la flecha agregamos el **contenido** de la función, que es lo que se va a **retornar**.

Funciones | Función flecha (arrow Function)

Si existe más de un parámetro, hay que usar paréntesis:

```
function sumar (num1,num2) {
    return num1+num2
}
console.log(sumar(4,6))
```

```
var aSumar = (num1,num2) => num1+num2
console.log(aSumar(5,7))
```

Mantenemos los parámetros entre paréntesis y colocamos a la derecha lo que devolverá la función.

```
// Función tradicional
function multiplicar (num1,num2) {
    producto= num1*num2
    return producto
}
console.log(multiplicar(2,3))
```

Función flecha de varias líneas.

```
// Función Arrow
var aMultiplicar = (num1,num2) =>
{
    producto= num1*num2
    return producto
}
console.log(aMultiplicar(6,7))
```

Funciones | Función flecha (arrow Function)

Existen varias formas de declarar una **función flecha**. Cada paso a lo largo del camino es una **función flecha** válida:

```
function (a){ // Función tradicional
  return a + 100
}

// Desglose de la función flecha
// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y las llaves de apertura.
(a) => { return a + 100 }

// 2. Quita los llaves{} del cuerpo y la palabra "return" (el return está implícito).
(a) => a + 100

// 3. Suprime los paréntesis de los argumentos
a => a + 100
```

Funciones | Función flecha - Sintaxis básica

```
//Un parámetro. Con una expresión simple no se necesita return:
param => expression

//Varios parámetros requieren paréntesis. Con una expresión simple no se necesita return:
(param1, paramN) => expression

//Las declaraciones de varias líneas requieren llaves y return:
param => {
  let a = 1
  return a + b
}

//Varios parámetros requieren paréntesis. Las declaraciones de varias líneas requieren llaves
y return:
(param1, paramN) => {
  let a = 1
  return a + b
}
```

Funciones | Función anónima

Las **funciones anónimas** son un tipo de funciones que se declaran sin nombre de función y se alojan en el interior de una variable y haciendo referencia a ella cada vez que queramos utilizarla:

En la consola mostramos el contenido de la variable (*sin ejecutarla, no hay paréntesis*) y nos devuelve la función en sí.
Luego **ejecutamos la función** contenida en la variable.

```
// Función anónima "saludo"
const saludo = function () {
    return "Hola"
  }
```

```
> saludo
ƒ () {
    return "Hola";
  }
> saludo()
'Hola'
>
```

Las **funciones anónimas** también permiten utilizar parámetros

```
// Función anónima "saludo"
const saludo = function (nombre) {
    var mensaje = "Hola " + nombre
    return mensaje
  }
```

```
> saludo("Luis")
'Hola Luis'
>
```

Scope (alcance)

El **scope** (alcance) determina la accesibilidad (visibilidad) de las variables. Define *¿en qué contexto las variables son visibles y cuándo no lo son?*. Una variable que no está "*al alcance actual*" no está disponible para su uso.

En JavaScript hay dos tipos de alcance:

- Alcance local (por ejemplo, una función)
- Alcance global (entorno completo de JavaScript)

Las variables definidas dentro de una función **no son accesibles** (visibles) desde fuera. La función "*crea un ámbito cerrado*" que impide el acceso a una variable de su interior desde fuera de ella o desde otras funciones.

Scope (alcance) | Variables locales

En el siguiente ejemplo creamos una variable llamada *carName* a la cual le asignamos un valor:

```
// aca no puedo usar la variable carName
function myFunction() {
    var carName = "Volvo"
    // aca si puedo usar la variable carName
}
// aca no puedo usar la variable carName
```

Podremos acceder al contenido de la variable **carName** solamente dentro de la función.

Este tipo de variables son de alcance local, porque solamente valen en el ámbito de la función, y no en el ámbito a nivel de programa. Los parámetros de la función funcionan como **variables locales** dentro de las mismas.

Scope (alcance) | Variables globales

Una variable declarada fuera de una función se convierte en **global**. Esto quiere decir que tiene alcance global: todos los scripts y funciones de una página web pueden acceder a ella.

```
var carName2 = "Fiat"
    // aqui si puedo usar carName2
function myFunction() {
    // aqui tambien puedo usar la variable carName2
}
```

En este caso podremos acceder al contenido la variable **carName** tanto desde fuera como desde adentro de la función

El alcance determina la accesibilidad de variables, objetos y funciones de diferentes partes del código.

Scope (alcance) | Variable automáticamente global

Si asignamos un valor a una variable que no ha sido declarada, **se convertirá en una variable global**. Este ejemplo declara la variable global **carName**, aún cuando su valor se asigna dentro de una función.

```
myFunction();
// aquí puede se puede usar carName
function myFunction() {
    carName = "Volvo"  // variable no declarada
}
```

En este caso podremos acceder al contenido la variable **carName** tanto desde fuera como desde adentro de la función por ser automáticamente global.

La vida útil de una variable comienza cuando se declara. Las variables locales se eliminan cuando se completa la función.

let y var

let declara una variable de alcance local, limitando su alcance (scope) al **bloque**, declaración, o expresión donde se está usando. **var** define una variable global o local en una función *sin importar el ámbito del bloque*.

```
var a = 5
var b = 10
if (a === 5) {
  let a = 4 // El alcance es dentro del bloque if
  var b = 15 // El alcance es global, sobreescribe a 10
  console.log(a)  // 4, por alcance a nivel de bloque
  console.log(b)  // 15, por alcance global
}
console.log(a) // 5, por alcance global
console.log(b) // 15, por alcance global
```

Callbacks (devolución de llamada)

Las funciones en JavaScript son objetos. Como cualquier otro objeto, se pueden pasar como parámetro. Por lo tanto, **podemos pasar una función como argumento de otra función**. Esto se llama función de devolución de llamada (**callback**). Las funciones también se pueden devolver como resultado de otra función.

```
function saludar(nombre) {
  alert('Hola ' + nombre)
}
function procesarEntradaUsuario(callback) {
  var nombre = prompt('Por favor ingresa tu nombre.')
  callback(nombre)
}
procesarEntradaUsuario(saludar)
```

El ejemplo es un callback sincrónico, ya que se ejecuta inmediatamente.

Clausuras (clousure)

Una **clausura** o cierre se define como una función que «encierra» variables en su propio ámbito (y que continúan existiendo aún habiendo terminado la función).

Es decir, devolvemos una función en una función externa que hace referencia a las variables locales de la función externa. Esto es posible si tenemos funciones anidadas en otra función y devueltas como referencia.

En la función interna, podemos usar las variables de la función externa. Debido al alcance de las variables locales, las funciones internas pueden acceder a las variables de la función externa.

Cuando devolvemos la función interna en la función externa, las referencias a las variables locales de la función externa todavía están referenciadas en la función interna.

Clausuras (clousure)

La función **iniciar()** crea una variable local llamada **nombre** y una función interna llamada **mostrarNombre()**. Por ser una función interna, esta última solo está disponible dentro de **iniciar()**. **mostrarNombre()** no tiene ninguna variable propia; pero puede acceder a la variable nombre declarada en la función iniciar().

```
function iniciar() {
  var nombre = "Codo a Codo" // La variable nombre es una variable local creada por iniciar.
  function mostrarNombre() { // La función mostrarNombre es una función interna, una clausura.
    alert(nombre); // Usa una variable declarada en la función externa.
  }
  mostrarNombre()
}
iniciar()
```

Clausuras | Ejemplo

La función creaSumador(x) toma un argumento único x y devuelve una nueva función. Esa nueva función toma un único argumento y, devolviendo la suma de x + y.

creaSumador es una fábrica de función. suma5 y suma10 son ambos closures. Comparten la misma definición de cuerpo de función, pero almacenan diferentes entornos. En el entorno suma5, x es 5. En lo que respecta a suma10, x es 10.

```
function creaSumador(x) {
    return function(y) {
      return x + y;
    };
  }

  var suma5 = creaSumador(5);
  var suma10 = creaSumador(10);

  console.log(suma5(2));  // muestra 7
  console.log(suma10(2)); // muestra 12
```

Material extra

Artículos de interés

Material de lectura:

- Funciones básicas
- Fundamentos sobre funciones
- Uso de la instrucción let en Javascript
- Funciones flecha, en Mozilla y W3Schools
- Callbacks y Clausuras

Videos:

- Curso Básico de Javascript - Funciones
- Funciones Arrow (de Flecha) Javascript 2018
- Qué es una función de flecha - JavaScript Arrow Functions
- ¿Qué es un callback en JavaScript?

Actividades prácticas:

- Del archivo “**Actividad Práctica - JavaScript Unidad 2**” están en condiciones de hacer los ejercicios: 1 a 18

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto