

JAVASCRIPT

Capítulo 4

Lenguajes de Marcas y Sistemas de Gestión de Información

Curso 2019/2020



Capítulo 4: PROGRAMACIÓN AVANZADA

1. Introducción
2. Funciones
3. Ámbito de las Variables
4. Sentencias `break` y `continue`
5. Otras Estructuras de Control



1. INTRODUCCIÓN

Las estructuras de control, los operadores y todas las utilidades propias de JavaScript que se han visto en los capítulos anteriores, permiten crear **scripts sencillos** y de **mediana complejidad**.

Sin embargo, para las aplicaciones más complejas son necesarios otros elementos como las **funciones** y otras **estructuras de control** más **avanzadas**, que se describen en este capítulo.



2. FUNCIONES

Cuando una serie de instrucciones **se repiten una y otra vez**, se complica demasiado el código fuente de la aplicación::

- El código de la aplicación es mucho más **largo** porque muchas instrucciones están repetidas.
- Si se quiere modificar alguna de las instrucciones repetidas, se deben hacer **tantas modificaciones como veces se haya escrito esa instrucción**, lo que se convierte en un trabajo muy pesado y muy propenso a cometer errores.

Las funciones son la solución a todos estos problemas. Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente.



2. FUNCIONES

```
var resultado;
```

```
var numero1 = 3;
```

```
var numero2 = 5;
```

```
// Se suman los números y se muestra el resultado
```

```
resultado = numero1 + numero2;  
alert("El resultado es " + resultado);
```

```
numero1 = 10;
```

```
numero2 = 7;
```

```
// Se suman los números y se muestra el resultado
```

```
resultado = numero1 + numero2;  
alert("El resultado es " + resultado);
```

```
numero1 = 5;
```

```
numero2 = 8;
```

```
// Se suman los números y se muestra el resultado
```

```
resultado = numero1 + numero2;  
alert("El resultado es " + resultado);
```

```
...
```



2. FUNCIONES

Agrupamos las **instrucciones comunes** en una función a la que se le puedan indicar los números que debe sumar antes de mostrar el mensaje.

Por lo tanto, en primer lugar se debe crear la función básica con las instrucciones comunes. Las funciones en JavaScript se definen mediante la palabra reservada `function`, seguida del nombre de la función. Su definición formal es la siguiente:

```
function nombre_funcion() {  
  
    ...  
  
}
```



2. FUNCIONES

```
resultado = numero1 + numero2;  
alert("El resultado es " + resultado);
```

Volviendo al ejemplo anterior, se crea una función llamada `suma` de la siguiente forma:

```
function suma() {
```

```
    resultado = numero1 + numero2;  
    alert("El resultado es " + resultado);
```

```
}
```



2. FUNCIONES

```
function suma() {  
    resultado = numero1 + numero2;  
    alert("El resultado es " + resultado);  
}
```

```
var resultado;  
var numero1 = 3;  
var numero2 = 5;
```

```
suma () ;
```

```
numero1 = 10;  
numero2 = 7;
```

```
suma () ;
```

```
numero1 = 5;  
numero2 = 8;
```

```
suma () ;
```

```
...
```




2. 1. ARGUMENTOS Y VALORES DE RETORNO

La mayoría de funciones de las aplicaciones reales deben **acceder** al valor de algunas **variables** para producir sus resultados.

Las variables que necesitan las funciones se llaman **argumentos**. Antes de que pueda utilizarlos, la función debe indicar **cuántos argumentos** necesita y cuál es el **nombre** de cada argumento. Además, al invocar la función, se deben incluir los valores que se le van a pasar a la función.



2. 1. ARGUMENTOS Y VALORES DE RETORNO

Siguiendo el ejemplo anterior, la función debe indicar que necesita dos argumentos:

```
function suma(primerNumero, segundoNumero) { ... }
```

Para utilizar el valor de los argumentos dentro de la función, se debe emplear el mismo nombre con el que se definieron:

```
function suma (primerNumero, segundoNumero) {  
    var resultado = primerNumero + segundoNumero;  
    alert("El resultado es " + resultado);  
}
```



2. 1. ARGUMENTOS Y VALORES DE RETORNO

- El número de argumentos que se pasa a una función **debería ser el mismo** que el número de argumentos que ha indicado la función. No obstante, **JavaScript no muestra ningún error si se pasan más o menos argumentos.**
- El **orden de los argumentos es fundamental**: el primer dato que se indica en la llamada, será el primer valor que espera la función...
- Se puede utilizar un **número ilimitado de argumentos**, aunque si su número es muy grande, se complica en exceso la llamada a la función.
- **No es obligatorio que coincida** el nombre de los argumentos que utiliza la función y el nombre de los argumentos que se le pasan. En el ejemplo anterior, los argumentos podrían ser `numero1` y `numero2` y los argumentos que utiliza la función son `primerNumero` y `segundoNumero`.



2. 1. ARGUMENTOS Y VALORES DE RETORNO

Ejemplo de una función que calcula el **precio total de un producto** a partir de su precio básico:

```
// Definición de la función

function calculaPrecio(precio) {

    var impuestos = 1.21;

    var gastosEnvio = 8;

    var precioFinal = ( precio * impuestos ) + gastosEnvio;

}

// Llamada a la función

calculaPrecio(22.50);
```



2. 1. ARGUMENTOS Y VALORES DE RETORNO

Lo ideal sería que la función pudiera **devolver el resultado** obtenido para guardarlo en otra variable y poder seguir trabajando con el:

```
// Definición de la función

function calculaPrecio(precio) {

    var impuestos = 1.21;

    var gastosEnvio = 8;

    var precioFinal = ( precio * impuestos ) + gastosEnvio;

}


// Llamada a la función

var precioTotal = calculaPrecio(22.50);
```



2. 1. ARGUMENTOS Y VALORES DE RETORNO

Las funciones no solamente puede recibir variables y datos, sino que también pueden **devolver** los valores que han calculado.

Para devolver valores dentro de una función, se utiliza la palabra reservada `return`. Aunque las funciones pueden devolver valores de cualquier tipo, solamente pueden devolver **un valor cada vez que se ejecutan**.



2. 1. ARGUMENTOS Y VALORES DE RETORNO

// Definición de la función

```
function calculaPrecio(precio) {  
    var impuestos = 1.21;  
    var gastosEnvio = 8;  
    var precioFinal = ( precio * impuestos ) + gastosEnvio;  
    return precioCalculado;  
}
```


// Llamada a la función

```
var precioTotal = calculaPrecio(22.50);
```

2. 2. ÁMBITO DE LAS VARIABLES

El ámbito de una variable (*scope*) es la zona del programa en la que se define la variable. JavaScript define dos ámbitos para las variables: **global** y **local**.

```
function creaMensaje() {  
    var mensaje = "Mensaje de prueba";  
}
```



A red oval highlights the variable declaration `var mensaje = "Mensaje de prueba";` inside the function. A red arrow points from this oval to the text **VARIABLE LOCAL**.

```
creaMensaje();  
  
alert(mensaje);
```

Se define en primer lugar una función llamada `creaMensaje` que crea una variable llamada `mensaje`. A continuación, se ejecuta la función mediante la llamada `creaMensaje()`; y seguidamente, se muestra mediante la función `alert()` el valor de una variable llamada `mensaje`.

2. 2. ÁMBITO DE LAS VARIABLES

Para mostrar el mensaje en el código anterior, la función `alert()` debe llamarse **desde dentro** de la función `creaMensaje()`:

```
function creaMensaje() {  
    var mensaje = "Mensaje de prueba";  
  
    alert(mensaje);  
}  
  
creaMensaje();
```



2. 2. ÁMBITO DE LAS VARIABLES

Existe el concepto de **variable global**, que está definida en cualquier punto del programa.

```
var mensaje = "Mensaje de prueba";  
  
function creaMensaje() {  
  
    alert(mensaje);  
  
}
```

Dentro de la función `creaMensaje()` se quiere hacer uso de la variable `mensaje` y que no ha sido definida dentro de la propia función → Si se ejecuta el código anterior, sí que se muestra el mensaje definido por la variable `mensaje`.

La variable `mensaje` se ha **definido fuera de cualquier función**. Este tipo de variables automáticamente se transforman en **variables globales** y están disponibles en cualquier punto del programa.

2. 2. ÁMBITO DE LAS VARIABLES

Si una variable se declara **fuera** de cualquier función, automáticamente se transforma en **variable global** independientemente de si se define utilizando la palabra reservada `var` o no.

Sin embargo, las variables definidas **dentro** de una función pueden ser **globales o locales**.

- Si en el **interior** de una función, las variables se declaran mediante `var` se consideran **LOCALES**.
- Las variables que **no se han declarado** mediante `var`, se transforman automáticamente en variables **GLOBALES**.



2. 2. ÁMBITO DE LAS VARIABLES: CURIOSIDADES

¿Qué sucede si una función define una variable local con el mismo nombre que una variable global que ya existe? En este caso, **las variables locales prevalecen sobre las globales, pero sólo dentro de la función:**

```
var mensaje = "gana la de fuera";
```

```
function muestraMensaje() {  
    var mensaje = "gana la de dentro";  
    alert(mensaje);  
}
```

```
alert(mensaje);  
muestraMensaje();  
alert(mensaje);
```



2. 2. ÁMBITO DE LAS VARIABLES: CURIOSIDADES

¿Qué sucede si dentro de una función se define una variable global con el mismo nombre que otra variable global que ya existe? En este otro caso, **la variable global definida dentro de la función simplemente modifica el valor de la variable global definida anteriormente:**

```
var mensaje = "gana la de fuera";  
  
function muestraMensaje() {  
    mensaje = "gana la de dentro";  
    alert(mensaje);  
}  
  
alert(mensaje);  
muestraMensaje();  
alert(mensaje);
```



3. SENTENCIAS `break` Y `continue`

Las sentencias `break` y `continue` permiten manipular el comportamiento normal de los bucles `for` para detener el bucle o para saltarse algunas repeticiones.

- La sentencia `break` permite terminar de **forma abrupta** un bucle.
- La sentencia `continue` permite **saltarse** algunas repeticiones del bucle.

+ 3.1. SENTENCIA break

El siguiente ejemplo muestra el uso de la sentencia `break`:

```
var cadena = "En un lugar de la Mancha de cuyo nombre no quiero acordarme...";  
var letras = cadena.split("");  
var resultado = "";  
  
for(i in letras) {  
    if(letras[i] == 'a') {  
        break;  
    }  
    else {  
        resultado += letras[i];  
    }  
}  
  
alert(resultado);  
  
// muestra "En un lug"
```

+ 3.1. SENTENCIA `break`

La utilidad de `break` es **terminar la ejecución** del bucle cuando una variable toma un determinado valor o cuando se cumple alguna condición.

En ocasiones, lo que se desea es **saltarse** alguna repetición del bucle cuando se dan algunas condiciones.



3.2. SENTENCIA continue

El siguiente ejemplo muestra el uso de la sentencia `continue`:

```
var cadena = "En un lugar de la Mancha de cuyo nombre no quiero acordarme...";  
var letras = cadena.split("");  
var resultado = "";  
  
for(i in letras) {  
    if(letras[i] == 'a') {  
        continue;  
    }  
    else {  
        resultado += letras[i];  
    }  
}  
  
alert(resultado);  
  
// muestra "En un lugr de l Mnch de cuyo nombre no quiero cordrm..."
```



4. OTRAS ESTRUCTURAS DE CONTROL

Las estructuras de control de flujo que se han visto (`if`, `else`, `for`) y las sentencias que modifican su comportamiento (`break`, `continue`) no son suficientes para realizar algunas tareas complejas y otro tipo de repeticiones.

Por ese motivo, JavaScript proporciona otras estructuras de control de flujo diferentes y en algunos casos más eficientes.

- `while`
- `do...while`
- `switch`

4.1. ESTRUCTURA `while`

Permite crear bucles que se ejecutan ninguna o más veces, dependiendo de la condición indicada:

```
while (condicion) {  
    ...  
}
```

“Mientras se cumpla la condición indicada, repite indefinidamente las instrucciones incluidas dentro del bucle”

- Si la condición **no se cumple**: el bucle no se ejecuta.
- Si la condición **se cumple**: se ejecutan las instrucciones una y otra vez hasta que la condición no se cumpla.



4.1. ESTRUCTURA `while`

Bucle `while` para sumar todos los números menores o iguales que otro número:

```
var resultado = 0;
var numero = 5;
var i = 0;

while(i <= numero) {
    resultado += i;
    i++;
}

alert(resultado); // resultado=1+2+3+4+5=15
```

4.2. ESTRUCTURA `do...while`

Es muy similar al bucle `while`, salvo que en este caso siempre **se ejecutan las instrucciones del bucle al menos la primera vez**:

```
do {  
    ...  
} while (condicion);
```

De esta forma, como la condición se comprueba después de cada repetición, la primera vez siempre se ejecutan las instrucciones del bucle. Es importante no olvidar que después del `while()` se debe añadir el **carácter ;**



4.2. ESTRUCTURA do...while

Calcular fácilmente el factorial de un número:

```
var resultado = 1;
```

```
var numero = 5;
```

```
do {
```

```
    resultado *= numero;    // resultado=resultado*numero
```

```
    numero--;
```

```
} while(numero > 0);
```

```
alert(resultado); // resultado=5*4*3*2*1=120
```



4.3. ESTRUCTURA switch

La estructura `if...else` se puede utilizar para realizar comprobaciones múltiples y tomar decisiones complejas.

```
if(numero == 5) {  
    ...  
}  
else if(numero == 8) {  
    ...  
}  
else if(numero == 20) {  
    ...  
}  
else {  
    ...  
}
```

¡¡ Si todas las condiciones dependen siempre de la misma variable, el código JavaScript resultante es demasiado redundante !!



4.3. ESTRUCTURA `switch`

La estructura `switch` es la más eficiente, ya que está especialmente diseñada para manejar de forma sencilla múltiples condiciones sobre la misma variable:

```
switch(variable) {  
    case valor_1:  
        ...  
        break;  
    case valor_2:  
        ...  
        break;  
    ...  
    case valor_n:  
        ...  
        break;  
    default:  
        ...  
        break;  
}
```




4.3. ESTRUCTURA `switch`

```
switch(numero) {  
    case 5:  
        ...  
        break;  
    case 8:  
        ...  
        break;  
    case 20:  
        ...  
        break;  
    default:  
        ...  
        break;  
}
```

¿Qué sucede si ningún valor de la variable del switch coincide con los valores definidos en los case? En este caso, se utiliza el valor `default` para indicar las instrucciones que se ejecutan en el caso en el que ningún case se cumpla para la variable indicada.

Aunque `default` es opcional, las estructuras `switch` suelen incluirlo para definir al menos un valor por defecto para alguna variable o para mostrar algún mensaje por pantalla.