Funciones en JavaScript

Una función es un bloque de código JavaScript que se define una vez pero que se puede ejecutar o llamar varias veces. Las funciones pueden tener **parámetros**, que son **variables locales** cuyo valor se especifica cuando se llama a la función.

Cuando se llama a una función en un objeto, la función se denomina **método** y el objeto en el que se invoca se pasa como argumento implícito de la función.

Ahora nos centraremos en ver cómo se definen y se llaman a nuestras propias funciones en JavaScript pero, sin olvidar, que JavaScript admite también "funciones integradas", como **eval()**, **parseInt()** y el método **sort()** de la clase *Array*. Además, JavaScript define otras funciones en el navegador, como **document.write()** y **alert()** tal y como iremos viendo a lo largo del curso.

Sumario

Definición y llamada de funciones

El método más fácil de definir una función es utilizar la instrucción **function**. Veamos unos ejemplos:

```
// Una función de acceso rápido,
//útil a veces en lugar de document.write()
// Esta función no tiene una instrucción return,
//por lo que devuelve undefined.
function print(msg) {
    document.write(msg, ?<br>?);
}

// Una función que calcula y devuelve
//la distancia comprendida entre dos puntos
function distancia(x1, y1, x2, y2) {
    let dx = x2 - x1;
    let dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}

//Una función recursiva (que se llama a sí misma)
//que calcula factoriales.
function factorial(x) {
    if (x <= 1) { return 1; }
    return x * factorial(x-1);
}
```

Con un código como el siguiente podemos llamar a las funciones definidas en el listado anterior:

```
print("Hola, " + nombre);
print("Hola caracola");
total_dist = distancia(0, 0, 2, 1) + distancia(2, 1, 3, 5);
print("La probabilidad es: " + factorial(5)/factorial(13));
```

Como JavaScript es un lenguaje de establecimiento flexible de tipos, no es necesario que se especifique un tipo de datos para los parámetros de la función y JavaScript no comprueba si hemos pasado el tipo de datos que espera la función. Si el tipo de datos de un argumento es importante, podemos comprobar dicho tipo con el operador **typeof**.

JavaScript tampoco comprueba si hemos pasado la cantidad correcta de argumentos. Si pasamos más argumentos de los esperados por la función, ésta ignora los adicionales. Si pasamos menos argumentos de los esperados, los parámetros que hemos omitido tienen un valor **undefined**.

> **Veremos a continuación cómo se puede determinar exactamente la cantidad de argumentos que se le pasa a una función y a acceder a dichos argumentos por su posición en la lista de argumentos en lugar de por su nombre.**

Denominación de funciones

Normas básicas:

- Los nombres de las funciones son normalmente verbos o frases que empiezan con verbos.
- Es normal que los nombres de función empiecen con letras minúsculas.
- Cuando un nombre incluye múltiples palabras, una convención utilizada con frecuencia es utilizar palabras separadas por guiones bajos, como en **guardar_documento()**.
- Otro convenio es empezar todas las palabras distintas con letras mayúsculas tras la primera palabra, como en **guardarDocumento()**.
- Las funciones que se supone que son internas o están ocultas, a veces tienen nombres que empiezan con un signo de subrayado **(_)**.

Instrucciones utilizadas en el interior de las funciones

return

La instrucción **return** especifica el valor devuelto por una función. La sintaxis de la instrucción **return** es:

```
return expresión;
```

Una instrucción **return** sólo puede aparecer dentro del cuerpo de una función. Será un error de sintaxis que aparezca en cualquier otro lugar.

Cuando se ejecuta la instrucción **return**, se evalúa la expresión y se devuelve como valor de la función y, claro está, se detiene la ejecución de dicha función, incluso aunque quede alguna otra instrucción en el cuerpo de la misma.

Veamos un ejemplo:

```
function cuadrado(x) { return x*x; }
```

Si una función ejecuta una instrucción **return** sin devolver nada, el valor de la expresión de llamada de la función es **undefined**.

throw

Una excepción es una señal que indica que se ha producido algún tipo de condición o error excepcional.

- Lanzar una excepción será señalar dicho error o condición excepcional.
- En el momento que se lanza una excepción la función termina indicando el error producido, a no ser que esa excepción sea "capturada".
- Capturar una excepción es controlarla, llevar a cabo cualquier acción necesaria o apropiada para recuperarse de la excepción.
- En JavaScript, las excepciones se lanzan siempre que se produce un error en tiempo de ejecución y cuando un programa lanza explícitamente una excepción utilizando la instrucción **throw**.
- Las excepciones se capturan con la instrucción **try**/**catch**/**finally**, tal y como veremos luego.

La instrucción **throw** tiene la siguiente sintaxis:

```
throw expresión;
```

La "expresión" puede ser "de cualquier tipo" pero, normalmente, es un objeto **Error** o una instancia de una de sus subclases (también veremos este objeto posteriormente). También es útil lanzar una cadena con un mensaje de error o un valor numérico (código de error). Veamos un ejemplo:

```
function factorial(x) {
        if (isNaN(parseInt(x)) || x < 0) throw new Error("Sólo números positivos");
        //Seguimos con el cálculo si son positivos
        for (var f = 1; x > 1; f *= x, x--); //Funcionaría esta función con "let f = 1..."?
        return f; }
```

Cuando se lanza una excepción, el intérprete de JavaScript detiene inmediatamente la ejecución normal del programa y salta al controlador de excepción más cercano. Los controladores de excepción se escriben utilizando la cláusula **catch** de la instrucción **try/catch/finally**, que luego se describe. Si no se encuentra ningún controlador de excepción, ésta se trata como un error y se informa al usuario.

try/catch/finally

La instrucción **try/catch/finally** es un mecanismo de control de excepciones de JavaScript. Veamos, a continuación, un código que ilustra la sintaxis y el propósito de la instrucción **try/catch**. En particular, hay que observar que a la palabra **catch** le sigue un identificador entre paréntesis. Este identificador es como un argumento de función. Denomina una variable local que existe sólo dentro del cuerpo del bloque **catch**. JavaScript asigna a esta variable cualquier objeto o valor de excepción.

```
//Función que recorre los elementos de un array
function factorial(x) {
    try {
        if (isNaN(parseInt(x)) || x < 0) throw new Error("Sólo números positivos");
        //Seguimos con el cálculo si son positivos
        for (var f = 1; x > 1; f *= x, x--); //Funcionaría esta función con "let f = 1..."?
        return f; }
    //Se guarda la excepción anterior en la variable local 'erro'
    catch(erro) { return erro; }
    //Y el bloque finally se ejecuta siempre
    finally { console.log("Este mensaje se ve siempre!!"); }
    }
    let n = prompt("Introduce número", "");
    alert("El factorial de " + n + " es " + factorial(n));
```

Este ejemplo es una instrucción **try/catch/finally**, aunque **finally** no se utiliza tan a menudo como **catch**, con frecuencia puede ser útil. Independientemente de cómo se complete el código en el bloque **try**, está garantizado que la cláusula **finally** se va a ejecutar si se ejecuta cualquier parte de dicho bloque. Generalmente se utiliza para limpiar tras el código de la cláusula **try**. Resumiendo, normalmente, el control llega al final del bloque **try** y, después, procede con el bloque **finally** que ejecuta cualquier limpieza necesaria.

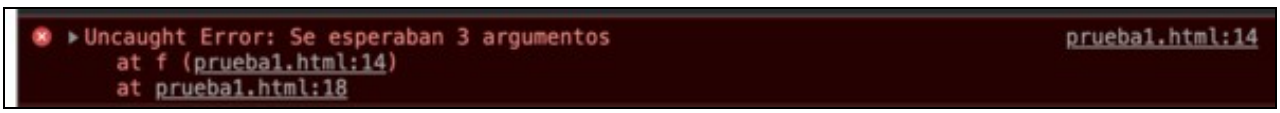
También saber que, si el control deja el bloque **try** por una instrucción **return**, **continue** o **break**, se ejecuta siempre el bloque **finally** antes de que el control se transfiera a su nuevo destino.

Argumentos de función

Los argumentos de una función se guardan en un objeto **Arguments** denominado **arguments**, parecido a una matriz, desde donde podremos acceder a cada uno de ellos por número (posición), no por nombre. El identificador **arguments**, como cualquier matriz, tiene la propiedad **length** que especifica el número de elementos que contiene.

En el siguiente ejemplo se demuestra el uso de **arguments** para verificar que se llama a una función con la cantidad correcta de argumentos, ya que JavaScript no lo hace automáticamente:

```
function f(x, y, z) {
    // Se comprueba número de argumentos
    if (arguments.length != 3) {
        throw new Error("Se esperaban 3 argumentos"); }
    // Aquí escribimos el cuerpo de la función?
}
```



Podemos escribir una función que trabaje con cualquier número de argumentos. Veamos un ejemplo en el que se crea una función **max()** que devuelve el valor del argumento más grande que se le pase (la función **Math.max()** se comporta de la misma forma):

```
function max() {
    let n = 0;
    let m = Number.NEGATIVE_INFINITY;
    // Recorrer con un bucle todos los argumentos
    //buscando y recordando el más grande
    for (let i = 0; i < arguments.length; i++) {
        n = parseInt(arguments[i]);
        if (n > m) m = n; }
    // Devolver el más grande
```

```
    return m;
}
let largest = max(1, 10, 23, 45, 33, 23, 11, 55, 2, 0);
console.log(largest);
```

Crear una función que devuelva la suma de todos esos argumentos.

```
function sumAll() {
    let sum = 0;
    for (let i = 0; i < arguments.length; i++) {
        sum += parseInt(arguments[i]);
    }
    return sum;
}
let suma = sumAll(1, 10, 23, 45, 33, 23, 11, 55, 2, 0);
console.log (suma);
```

Paso de múltiples argumentos

Cuando una función requiere más de tres argumentos, es interesante poder hacer que los argumentos se pasen como pares nombre/valor y en cualquier orden. Para implementar este tipo de invocación de método, debemos definir nuestra función para que espere un solo objeto como su argumento y, posteriormente, dejar que los usuarios de la función pasen un literal de objeto que defina los pares de nombre/valor requeridos.

Nos pararemos en estas funciones cuando veamos Objetos en un apartado posterior.

Funciones como datos

En JavaScript, las funciones no son sólo sintaxis sino también datos, lo que significa que se pueden asignar a variables, guardarse en las propiedades de los objetos o de los elementos de matrices, pasarse como argumentos a funciones, etc.

```
function square(x) { return x*x; }
let a = square(4) // "a" contiene el número 16
let b = square;  // "b" hace referencia a la misma función square
let c = b(5); // "c" contiene el número 25
```

Las funciones también se pueden asignar a propiedades de un objeto en lugar de a variables globlales. Así las funciones se denominan métodos:

```
let o = new Object;
o.square = function(x) { return x*x; }
let y = o.square(16);  // y es igual a 256
```

Las funciones ni siquiera requieren un nombre si se asignan a elementos de una matriz:

```
let a = new Array(3);
a[0] = function(x) { return x*x; }
a[1] = 20;
a[2] = a[0](a[1]);//"a[2]" contiene el número 400
```

Podemos ver en los dos ejemplos siguientes cómo se pueden pasar funciones como argumentos de otras funciones y lo útil que puede ser esta técnica a la hora de programar en JavaScript. Ejemplo simple:

```
//Definimos algunas funciones simples
function sumar(x,y) { return x + y; }
function restar(x,y) { return x - y; }
function multiplicar(x,y) { return x * y; }
function dividir(x,y) { return x / y; }
//Definimos una función que acepta como argumento
//una de las definidas antes
function operar(operador, operando1, operando2) {
    return operador(operando1, operando2);
}
//Solucionar la ecuación : (2+3) + (4*5)
let i = operar(sumar, operar(sumar,2,3), operar(multiplicar,4,5));
//Mostramos la salida
console.log(i);
```

Ejemplo más complejo:

```
//Definimos algunas funciones simples
//como elementos de un objeto diccionario
let operadores = {
    sumar : function sumar(x,y) { return x + y; },
    restar : function restar(x,y) { return x - y; },
    multiplicar : function multiplicar(x,y) { return x * y; },
    dividir : function dividir(x,y) { return x / y; },
    cuadrado : Math.pow //También con predefinidas
}
//Función que llama a una función desde su operador
function operar(nombreOperador, operando1, operando2) {
    if (typeof operadores[nombreOperador] == "function")
        return operadores[nombreOperador](operando1, operando2);
    else throw "Operador NO conocido";
}

//Ejemplo de llamada de esta función: (2+3) + (4*5)
let i = operar("sumar", operar("sumar",2,3), operar("multiplicar",4,5));
//Mostramos la salida
console.log(i);
```

Funciones anónimas

Son funciones sin un nombre o identificador. Debido a esto, se pueden pasar a otras funciones o asignar a variables. Cuando una función anónima se asigna a una variable, el nombre de la variable es el que usamos para llamar a la función.

Una buena explicación la encontramos en este enlace.

> **Las funciones anónimas son extremadamente útiles porque nos permiten definir complicados patrones de programación, necesarios para construir aplicaciones profesionales. Las utilizaremos en el apartado Programación Asíncrona en JavaScript.**

Funciones estándar

Son funciones definidas por JavaScript. Estas funciones realizan procesos que simplifican tareas complejas. Las siguientes son las que más se usan:

- **isNaN(valor)** ? Devuelve true si el valor entre paréntesis no es un número.
- **parseInt(valor)** ? Convierte una cadena de caracteres con un número en un número entero.
- **parseFloat(valor)** ? Convierte una cadena de caracteres con un número en un número decimal.
- **encodeURIComponent(valor)** ? Codifica una cadena de carateres para ser incluida en una URL.
- **decodeURIComponent(valor)** ? Decodifica una cadena de caracteres.

Podemos ver un ejemplo donde se ve el funcionamiento de **encodeURIComponen()**.

```
let nombre = "Juan Pérez"
let codificado = encodeURIComponent(nombre);
let miURL = "http://www.ejemplo.com/contacto.html?nombre=" + codificado;
console.log(miURL)
//salida:http://www.ejemplo.com/contacto.html?nombre=Juan%20P%C3%A9rez
```

Recursividad

Está perfectamente bien que una función se llame a sí misma, siempre que no lo haga con tanta frecuencia que desborde la pila. Una función que se llama a sí misma se llama recursiva. La recursividad permite escribir algunas funciones con un estilo diferente.

Veamos un ejemplo con una función que devuelve la potencia de un número elevado a otro: $2^5 = 2 * 2 * 2 * 2 * 2 = 32$

- Función **power(base, exponente)** programada utilizando la sentencia **for**:

```
function power(base, exponente) {
    let result = 1;
    for (let cont = 0; cont < exponente; cont++) {
        result *= base;
    }
    return result;
}
```

- Función **power(base, exponente)** programada utilizando **recursividad**:

```
function power(base, exponente) {
    if (exponente == 0) {
        return 1;
    } else {
        return base * power(base, exponente - 1);
    }
}
```

Pero, esta implementación tiene un problema pues es unas tres veces más lenta que la versión del *for loop*.

Aquí aparece el dilema de la elegancia frente a la velocidad de procesamiento. Y, preocuparse sobre la eficiencia, en muchas situaciones puede ser simplemente una distracción.

Pero, la recursividad no siempre es, simplemente, una ineficiente alternativa a los bucles. Por ejemplo, podemos considerar esta situación: "Comenzando desde el número 1 y, repetidamente, sumar 5 o multiplicar por 3, se puede producir un conjunto infinito de números. ¿Cómo escribirías una función que, dado un número, intenta encontrar una secuencia de tales sumas y multiplicaciones que produzca ese número?"

(Por ejemplo, para el número 13 = 1 * 3 + 5 + 5. Pero, para el número 15 no existe ningún modo de conseguirlo con este método).

Podemos ver una solución recursiva:

```
function encontrarSolucion(objetivo) {
    function encontrar(actual, historial) {
        if (actual == objetivo) {
            return historial;
        } else if (actual > objetivo) {
            return null;
        } else {
            return encontrar(actual + 5, `(${historial} + 5)`) || encontrar(actual * 3, `${historial} * 3`);
        }
    }
    return encontrar(1, "1");
}
```

Volver