

Desarrollo Web en entorno Cliente: JAVASCRIPT

UNIDAD 3. FUNCIONES

3.1 FUNCIONES

3.2 FUNCIONES PROPIAS DEL LENGUAJE

[parseInt\(\)](#)
[parseFloat\(\)](#)

[escape\(\)](#) → [encodeURIComponent\(\)](#) y [encodeURIComponent\(\)](#)
[unescape\(\)](#) → [decodeURI\(\)](#) y [decodeURIComponent\(\)](#)

[isNaN\(\)](#)
[eval\(\)](#)

3.3 FUNCIONES SIN PARÁMETROS

3.4 PARÁMETROS POR VALOR Y PARÁMETROS POR ¿REFERENCIA?

3.5 ARGUMENTOS VARIABLES

3.6 PARÁMETROS POR DEFECTO

3.7 FUNCIONES RECURSIVAS

3.8 LAS FUNCIONES SON DATOS

3.9 FUNCIONES ANÓNIMAS, AUTOEJECUTABLES Y QUE DEVUELVEN FUNCIONES

3.10 FUNCIONES INTERNAS O DEFINIDAS DENTRO DE OTRA FUNCIÓN

UNIDAD 3. FUNCIONES

3.1 FUNCIONES

¿Qué hay en una función?

Una función en javascript es un conjunto de declaraciones que pueden ser invocadas desde cualquier parte de un programa javascript.

Las funciones pueden aceptar argumentos o valores introducidos en ellas. Dentro de una función se puede actuar sobre esos argumentos y devolver valores usando return.

Las funciones son perfectas cuando tenemos algo que tiene que ocurrir varias veces dentro de un programa, por ejemplo una función creada para verificar que determinados campos imprescindibles de un formulario han sido rellenados.

Una función se define mediante la palabra reservada function, seguida del nombre de la función y después entre paréntesis los argumentos opcionales o parámetros para usar en ella. Las llaves encierran las declaraciones a ejecutar como parte de la función:

```
function nombre_Funcion (arg1,arg2, ... , argN){    //puede que la función no tenga argumentos
                                                    // solo (), argumentos opcional

    //cuerpo de la función, código de la función
    return Valor;    // Devuelve valor de la función, es opcional
}
```

```
function nombre_Funcion () {
    //cuerpo de la función, código de la función
}
```

○ El nombre de la función debe cumplir los siguientes requisitos:

- Deben usarse sólo letras, números o el caracter de subrayado.
- Debe ser único en el código JavaScript de la página web.
- No pueden empezar por un número.
- No puede ser una de las palabras clave o reservadas del lenguaje.

NOTA: Cuando se define una función, es importante observar que el código de la función no se ejecuta realmente hasta que la función es llamada o invocada. Se suelen definir en <head></head>, pero también se pueden definir en <body></body> (por esto el comentario anterior), siempre sin olvidar que tiene que ir entre <script type=" " "> </script>.

Podremos llamar a nuestra función (definida en la cabecera head del documento HTML) desde cualquier parte del código y también como respuesta a algún evento, por ejemplo:

```
<a href="doc.htm" onClick="javascript:NombreFuncion(arg1,...)">Pulse  
por aquí, si es tan amable</a>
```

No es obligatorio poner dentro del evento: javascript:... Funciona con y sin.

Ejemplo –Función que calcula el importe de un producto después de haberle aplicado el IVA:

```
function aplicar_IVA(valorProducto, IVA){  
    var productoConIVA= (valorProducto* IVA/100)+valorProducto;  
    alert("El precio del producto, aplicando el IVA  
        del " + IVA + "% es: " + productoConIVA);  
}
```

La invocaremos pasándole dos valores `aplicar_IVA(300, 1.18)`.

¿Si llamamos a la función `aplicar_IVA(1.18, 300)` el resultado será el mismo?

Ejemplo anterior: [4.8-AplicarIVA_b.html](#) [4.7-AplicarIVA_a.html](#)

Invocar una función desde JavaScript:

```
<html>  
  <head>  
    <title>InvocarfuncióndesdeJavaScript</title>  
    <script type="text/javascript">  
      function mi_funcion([args]){  
        //instrucciones  
      }  
    </script>  
  </head>  
  <body>  
    <script type="text/javascript">  
      mi_funcion([args]);  
    </script>  
  </body>  
</html>
```

Dado el Ejemplo anterior: [4.9-InvocarFunciónDesdeJavaScript.html](#) Probar a invocar la función desde el mismo script o desde otro (en distinto orden antes y después de la definición de la función) y en <head> y en <body>.

Invocar una función desde HTML:

```
<html>  
  <head>  
    <title>InvocarfuncióndesdeJavaScript</title>  
    <script type="text/javascript">
```

```
        function mi_funcion([args]){
            //instrucciones
        }
    </script>
</head>
<body onload="mi_funcion([args])"></body>
</html>
```

Ejemplo anterior: [4.10-InvocarFunciónDesdeHTML.html](#)

RECORDAR: Alcance de las variables

Variables locales y globales. Se refiere a los lugares desde los cuales se puede acceder a su valor. Las variables tienen un alcance global cuando se definen y utilizan fuera de una función. En el contexto de una página web, o de un documento, una variable global es accesible y utilizada a lo largo de ese documento.

Variables locales son las definidas dentro de una función y solamente tienen el alcance de esa función. No se puede acceder a los valores de esa variable fuera de la función. Los parámetros de la función también se limitan localmente a la función.

Acceder al archivo donde definimos los ámbitos y Hoisting en las distintas formas de declarar variables: var (en ES5), let y const (en ES6)

[var_let_const_ambitos](#)

Por defecto las variables definidas en Javascript sin ponerles delante var tienen el mismo ámbito que si fuesen definidas con var.

Los argumentos de las funciones suelen ser nombres de variables y no deberían nombrarse igual que las variables que se usan para llamar o invocar la función. Esto se debe a que puede crear confusión en el código y en el alcance que tengan.

Lo recomendable es que se utilicen nombres diferentes para las variables dentro y fuera de las funciones, y utilizar siempre la palabra clave *var* para iniciar las variables.

Ejemplo: [scoping.htm](#)

3.2 Funciones propias del lenguaje

Las funciones que se van a describir aquí son funciones que no están ligadas a los objetos del navegador, sino que forman parte del lenguaje. Estas funciones nos van a permitir convertir cadenas a enteros o a reales, evaluar una expresión, ...

Las funciones que trataremos serán estas:

- [parseInt](#)
- [parseFloat](#)
- [escape](#) → [encodeURIComponent\(\)](#) y [encodeURIComponent\(\)](#)
- [unescape](#) → [decodeURI\(\)](#) y [decodeURIComponent\(\)](#)
- [isNaN](#)
- [eval](#)

parseInt(cadena, base)

Esta función devuelve un entero como resultado de convertir la cadena que se le pasa como primer argumento.

Opcionalmente, podemos indicarle la base en la que queremos que nos devuelva este entero (octal, hexadecimal, decimal, ...). Si no especificamos base, nos devuelve el resultado en base diez. Si la cadena no se puede convertir a un entero, la función devuelve el mayor entero que haya podido construir a partir de la cadena.

Si el primer caracter de la cadena no es un número, la función devuelve NaN (Not a Number).

```
a = parseInt("1234");
```

En 'a' tenemos almacenado el valor 1234, y podremos hacer cosas como a++, obteniendo 1235 ;)
Produce el mismo efecto que hacer

```
a = parseInt("1234",10);
```

puesto que, como hemos dicho, la base decimal es la base por defecto.

```
a = parseInt("123cuatro5");
```

En 'a' tenemos almacenado el valor 123.

```
a = parseInt("a1234");
```

En 'a' tenemos almacenado el valor NaN.

```
a = parseInt("FF",16);
```

En 'a' tenemos almacenado el valor 255.

Ejemplo:

```
<script type="text/javascript">  
  var input = prompt("Introduce un valor: ");  
  var inputParsed= parseInt(input);  
  alert("parseInt("+input+"): "+inputParsed);  
</script>
```

Ejercicio alumnado: probar este ejemplo usando todos los valores anteriores.

Ejemplo: [4.5-parseInt.html](#)

parseFloat(cadena)

Esta función nos devuelve un real como resultado de convertir la cadena que se le pasa como argumento.

Si la cadena no se puede convertir a un real, la función devuelve el mayor real que haya podido construir a partir de ella. Si el primer carácter no puede ser convertido en un número, la función devuelve NaN.

```
a = parseFloat("1.234");
```

En 'a' tenemos almacenado el valor 1.234.

```
a = parseFloat("1.234e-1");
```

En 'a' tenemos almacenado el valor 0.1234.

```
a = parseFloat("1.2e-1");
```

En 'a' tenemos almacenado el valor 0.12.

```
a = parseFloat("e");  
a = parseFloat("e-1");  
a = parseFloat(".e-1");
```

Todas ellas producen a = NaN.

```
a = parseFloat("1.e-1");
```

En este caso a = 0.1

Ejemplo: [4.6-parseFloat.html](#)

Nota: Las funciones `escape` y `unescape` no trabajan apropiadamente con caracteres no ASCII y han sido desaprobadadas. En JavaScript 1.5 y posteriores, utilizar `encodeURIComponent`, `decodeURI`, `encodeURIComponent` y `decodeURIComponent` (http://www.w3schools.com/jsref/jsref_encodeuri.asp)

isNaN(valor)

Esta función evalúa el argumento que se le pasa y devuelve 'true' en el caso de que el argumento NO sea numérico. En caso contrario (el argumento pasado es numérico) devuelve 'false'. Por ejemplo:

```
isNaN("Hola Mundo");  
    devuelve 'true', pero
```

```
isNaN("091");  
    devuelve 'false'.
```

Ejemplo: [4.3-isNaN.html](#)

eval(expresion)

La función eval convierte una expresión en código utilizable por Javascript.

El uso más común de eval() se hace en scripts que construyen poco a poco expresiones o sentencias utilizando varias cadenas. Por ejemplo, el script puede asignar distintos valores a variables dependiendo del navegador o de un valor introducido por el usuario. El script concatenará las cadenas y aplicará eval.

El siguiente programa es un ejemplo:

```
<html>  
<head></head>  
<body>  
<script language="javascript">  
    <!--  
    var str1="Number";  
    var str2;  
    if (confirm("Haga click en Aceptar para obtener el valor máximo;"  
        + " haga click en Cancelar para obtener el mínimo"))  
        str2="MAX_VALUE";  
    else  
        str2="MIN_VALUE";  
    document.write("<pre>");  
    document.write("Sin eval(): "+ str1 + "." +str2);  
    document.write("<br>");  
    document.write("Con eval(): "+ eval(str1 + "." +str2));  
    document.write("</pre>");  
    -->  
</script>  
</body>  
</html>
```

Ejemplo anterior (probarlo alumnado): [eval.html](#)

Otro Ejemplo (probarlo alumnado) [4.2-eval.html](#)

```
<script type="text/javascript">
```

```
var input = prompt("Introduce una operación numérica");
var resultado = eval(input);
alert("El resultado de la operación es: " + resultado);
</script>
```

3.3 Funciones sin parámetros

En el ejemplo siguiente se ha definido una función sin parámetros `Fecha_de_hoy()` que obtiene la fecha de la máquina del cliente y la escribe en el documento. Para realizar este trabajo se utiliza la clase que se encuentran disponible de manera nativa en Javascript llamada `Date`, de la que se puede obtener más información en el capítulo siguiente de objetos del lenguaje.

```
<html>
<head>
  <script language="javascript">
    <!--
    function fecha_de_hoy()
    {
      var fecha=new Date();

      document.write(fecha.getDate());
      document.write("/");
      document.write(fecha.getMonth()+1);
      document.write("/");
      document.write(fecha.getFullYear());
    }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
    document.write("Bienvenido, hoy es: ");
    fecha_de_hoy();
    -->
  </script>
</body>
</html>
```

Ejemplo anterior: [FechaDeHoy.html](#) (obtiene y visualiza la fecha del cliente)

3.4 Parámetros por valor y Parámetros por referencia?

Las funciones nos permiten pasar argumentos o parámetros para manipularlos en su interior. Son conocidas dos estrategias como el **paso por valor** y el **paso por referencia**.

<https://medium.com/laboratoria-developers/por-valor-vs-por-referencia-en-javascript-de3daf53a8b9>

JavaScript sólo permite el **paso por valor**. Cuando asignamos valores *primitivos* ([Boolean](#), [Null](#), [Undefined](#), [Number](#), [String](#) y [Symbol](#)), el valor asignado es una copia del valor que estamos asignando. Pero cuando asignamos valores *NO primitivos* o complejos ([Object](#), [Array](#) y [Function](#)), JavaScript copia “la referencia”, lo que implica que no se copia el valor en sí, si no una referencia a través de la cual accedemos al valor original.

Ver ejemplo: [Paso_parametros_simp_y_comp_Func_ulti.html](#)

Valor primitivo **Symbol**, novedad de ECS6, verlo en <https://www.oscarlijo.com/blog/symbol-en-javascript/>

```
const a = Symbol("Oscar");
const b = Symbol("Oscar");
console.log(a === b); // Devuelve false
```

En JavaScript, cuando asignamos un valor a una variable, o pasamos un argumento a una función, este proceso siempre se hace “por valor” (*by value* en inglés). Estrictamente hablando, JavaScript no nos ofrece la opción de pasar o asignar “por referencia” (*by reference* en inglés), como en otros lenguajes. Lo interesante en nuestro caso, es que cuando una variable hace referencia a un *objeto* ([Object](#), [Array](#) o [Function](#)), el “valor” es la referencia en sí.


Paso de argumentos por valor a una función de JavaScript

Para las funciones de JavaScript los argumentos declarados como **número** y **booleanos** serán considerados en **paso por valor**. Así lo que estamos pasando es realmente una copia de esa variable. Cualquier cambio de esa variable en el interior de la función no afecta al original. En este ejemplo pasamos un número y un booleano y los modificamos dentro:

```
<script>
  function modificaNumBool(unNumero, unBooleano) {
    unNumero = 9999;
    unBooleano = true;
  }
  var miNumero = 1;
  var miBooleano = false;
</script>
```

Paso de números y booleanos por valor:

```
<input type="button" value="modifica(unNumero, unBooleano)"
onclick = "modificaNumBool(miNumero, miBooleano);
  alert('miNumero = ' + miNumero + ', miBooleano = ' +
  miBooleano)" />
```

 Paso de números por valor: miNumero=1, miBooleano=false

Ejemplo anterior: [Paso_numeros_por_valor.html](#)

Se observa que tenemos una función que recibe unos argumentos y luego los modifica dentro de ella. Declaramos el número **1** y el booleano **false** y con la ejecución del botón llamamos a esa función para cambiar esas variables a **9999** y **true**, pero al final volvemos a obtener **1** y **false**. Sucede que unNumero y unBooleano son variables que JavaScript toma por defecto en **paso por valor** y por tanto sólo tiene un alcance interno en la función. Los cambios hechos en ella no afectan al original desde donde se copió.


El paso por valor ocasiona que Javascript deba crear memoria adicional para albergar las copias. Esto no es importante si lo que ocupan los valores no es significativo, pero en otro caso habrá que tenerlo en cuenta.

Paso de argumentos por ¿referencia? a una función de JavaScript

Por defecto, Javascript pasará los **objetos**, **arrays** y **funciones** por referencia. Pero esto no es del todo cierto y es por lo que le hemos puesto unos interrogantes a este título, como comprobaremos al finalizar este apartado.

En este primer ejemplo pasamos un **Object** de Javascript:

```
<script>
function modificaObjeto(unObjeto) {
    unObjeto.propiedad = "Valor MODIFICADO";
}
var miObjeto = new Object();
miObjeto.propiedad = "Valor original";
</script>
Paso de objetos por referencia:
<input type="button" value="SI modifica mis datos"
onclick = "modificaObjeto(miObjeto);
alert('miObjeto.propiedad=' + miObjeto.propiedad) " />
```

 Paso de objetos por referencia: miObjeto.propiedad= Valor MODIFICADO

Ejemplo anterior: [Paso_objetos_por_ref_mod.html](#)

La función recibe un objeto con una propiedad y modifica ésta en su interior. Luego con el alert obtenemos la propiedad modificada en el objeto **original**.

Sin embargo otra cosa es que modifiquemos la naturaleza del propio objeto, por ejemplo declarando dentro de la función un nuevo objeto en ese mismo argumento:

```
<script>
function noModificaObjeto(unObjeto) {
    unObjeto = new Object();
```

```

        unObjeto.propiedad = "Valor MODIFICADO";
    }
    var miObjetoB = new Object();
    miObjetoB.propiedad = "Valor original";
</script>
Paso de objetos por referencia:
<input type="button" value="NO modifica mis datos"
    onclick = "noModificaObjeto(miObjetoB);
    alert('miObjetoB.propiedad=' + miObjetoB.propiedad)" />

```

 Paso de objetos por referencia: miObjetoB.propiedad= Valor original

Ejemplo anterior: [Paso_objetos_por_ref_orig.html](#)


Ahora el objeto se vuelve a declarar en el interior de la función, con lo que pierde la referencia al objeto original, siendo ahora una **nueva variable** con alcance limitado a ese interior. Así el mensaje nos devuelve a la propiedad del objeto original, no la del modificado.

Esto también pasa con un **Array**:

```

<script>
    function modificaArray(unArray) {
        unArray.push(unArray.length+1);
    }
    function noModificaArray(unArray) {
        unArray = new Array("a", "b", "c");
    }
    var miArray = new Array(1, 2, 3);
</script>
Paso de arrays por referencia:
<input type="button" value="NO modifica mi array"
    onclick = "noModificaArray(miArray); alert(miArray)" />
<input type="button" value="SI modifica mi array"
    onclick = "modificaArray(miArray); alert(miArray)" />

```

 Paso de arrays por referencia: 1,2,3 (no modifica mi array)
1,2,3,4 (si modifica mi array)

Ejemplo anterior: [Paso_Arrays_por_ref.html](#)

¿Entonces los objetos se pasan por referencia o no?. Diríamos que sí y no. Esto se explica entendiendo que JavaScript **pasa todos los argumentos POR VALOR**. En el caso del paso de objetos lo que hace es copiar el puntero de la referencia al objeto, o dicho de otra forma, hace una copia del valor de ese puntero, que no es otra cosa que una dirección de memoria. Pero es una **copia de la referencia**, no la referencia original, por lo que cuando asignamos a ese puntero un nuevo objeto dentro de la función, entonces no se modifica el objeto original sino esa copia de puntero. Otra cosa es que, mientras no se modifique ese puntero, Javascript permite acceder a las propiedades del objeto que apunta y, por tanto, modificarlas.

Es una cuestión de terminología, pero se puede decir que Javascript no pasa por referencia en ningún caso. Sólo por valor para números y booleanos y con **copia de la referencia** para los objetos. A esta estrategia a veces también se le llama *call by object* o *call by object-sharing* (llamada por objeto o por objeto compartido), cuyo comportamiento según vimos es el siguiente:

- Permite acceder al objeto original, del que tenemos una copia de la referencia, para modificar sus propiedades y métodos. Este es un comportamiento "equivalente" al paso por referencia.
- Las nuevas asignaciones hechas a la referencia copiada no modifican el objeto original sino la copia, comportándose en este caso como un paso por valor.

Otros EJEMPLOS 1:

Las funciones en Javascript pueden devolver un valor utilizando la sentencia return.

```
<html>
<head>
  <script language="javascript">
    <!--
      function dia_semana(numero_de_dia)
      {
        var dia=new Array(7);
        dia[0]="domingo";
        dia[1]="lunes";
        dia[2]="martes";
        dia[3]="miercoles";
        dia[4]="jueves";
        dia[5]="viernes";
        dia[6]="sábado";

        return dia[numero_de_dia];
      }

      function fecha_de_hoy()
      {
        var fecha=new Date();

        document.write(dia_semana(fecha.getDay()));
        document.write(", ");
        document.write(fecha.getDate());
        document.write("/");
        document.write(fecha.getMonth()+1);
        document.write("/");
        document.write(fecha.getFullYear());
      }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
      document.write("Bienvenido, hoy es: ");
      fecha_de_hoy();
    -->
  </script>
</body>
</html>
```

```
-->
</script>
</body>
</html>
```

Ejemplo anterior: [FechaDeHoy2.html](#) (obtiene y visualiza la fecha del cliente incluyendo día de la semana)

Otros EJEMPLOS 2: (por ¿referencia?)

```
<html>
<head>
  <script language="javascript">
    <!--
    function cuadrado(numero)
    {
      numero=numero*numero;
      return numero;
    }

    function objeto_numero(valor)
    {
      this.numero=valor;
    }

    function cuadrado_objeto(objeto)
    {
      objeto.numero=objeto.numero*objeto.numero;
      return objeto.numero;
    }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
var numero1=4;
var numero2=new objeto_numero(4);

document.write("Cuadrado del número 4 como parámetro simple: ");
document.write(cuadrado(numero1));
document.write("<br>");
document.write("Valor de la variable simple después de" +
  " llamar a la función: ");
document.write(numero1);
document.write("<br>");
document.write("Cuadrado del número 4 como parámetro de tipo objeto: ");
document.write(cuadrado_objeto(numero2));
document.write("<br>");
document.write("Valor de la variable objeto después de" +
  " llamar a la función: ");
document.write(numero2.numero);
```

```
-->
</script>
</body>
</html>
```

Ejemplo anterior: [ParametroReferencia.html](#)

3.5 Argumentos variables

Las funciones en JavaScript tienen una propiedad particular, y es que no tienen un número fijo de argumentos. Es decir, nosotros podremos llamar a las funciones con un número cualquiera de argumentos y con cualquier tipo de argumentos.

Los argumentos pueden ser accedidos bien por su nombre, bien por un vector de argumentos que tiene asociada la función (que será NombreFuncion.arguments), y podemos saber cuántos argumentos se han enviado viendo el valor de NombreFuncion.arguments.length

Por ejemplo:

```
function Cuenta() {
    document.write("Me han llamado con " + Cuenta.arguments.length
        + " argumentos<br>");
    for(i = 0; i < Cuenta.arguments.length; i++)
        document.write("Argumento " + i + ": "
            + Cuenta.arguments[i] + "<br>");
}
```

La podemos llamar con:

```
<script language="JavaScript">
    <!--
        Cuenta("Hola", "a", "todos");
    -->
</script>
```

Ejercicio alumnado: Probar el código anterior.

El alumnado hará este ejemplo en los ejercicios [ParametrosVariables.html](#)

Operador rest ... de JavaScript ES6

El operador rest es una característica de JavaScript ES6 que nos permite evitar el trabajo repetitivo en las funciones con el tratamiento de parámetros. Gracias a ES6, podemos definir valores predeterminados a las funciones, aunque no solo se limita a esto, sino que el operador rest nos va a permitir obtener un número indefinido de parámetros de forma estructurada a través de un array de valores.

El uso de rest es bastante sencillo, ya que podemos emplearlo dentro del juego de parámetros definido en la cabecera de una función. Veámoslo con un ejemplo de código en el que indicaremos como parámetro de la función "...numeros":

```
function max(...numeros) {  
  console.log(numeros);  
}
```

En este ejemplo, lo que indicamos es que cualquier cantidad de parámetros que se envíen al invocar la función se estructurará como un array y dentro de la función se conocerá con el nombre de "numeros". De este modo, podremos invocar a la función con el número de parámetros que queramos, teniendo siempre en el parámetro *numero* un array, con el número de casillas deseado.

Para comprobar esto, basta con invocar a la función anterior utilizando el juego de parámetros que deseemos:

```
max(1, 2, 6);  
max();  
max(1, 5, 6, 7, 10001);
```

Podemos hacerlo así, o también mezclando varios tipos de parámetros:

```
max("test", 4, true, 2000, "90");
```

*****Los parámetros de tipo REST son una nueva funcionalidad añadida al lenguaje en la especificación ES6. Se aplican en funciones (tradicionales o anónimas) y permite recibir n número de parámetros.

```
function test(a,b, ...c){  
  //cuerpo de la función  
}  
// ...c es un parámetro de tipo REST
```

Los 3 puntos suspensivos dentro de la zona de parámetros de una función indican un parámetro de tipo REST

Siempre llegará un array

Es importante tener en cuenta que siempre que utilicemos el operador rest nos llegará un array como valor del parámetro. De este modo, tenemos la certeza de que podremos utilizar los métodos de recorrido y manipulación de los array. Antes de que tuviésemos el operador rest en JavaScript existía la posibilidad de obtener cualquier número de parámetros a través del objeto arguments disponible en la función. Pero, en ese caso, como arguments no era un array, no permitía algunos tipos de operaciones a menos de que lo transformásemos previamente en un array.

3.6 Parámetros por defecto

Como acabamos de ver, en Javascript se puede trabajar con un número de parámetros variable. Esta característica permite controlar desde la función la posibilidad de trabajar con valores por defecto, si no se pasan parámetros al invocarla.

```
<html>
<head>
  <script language="javascript">
    <!--
      function bienvenida(nombre)
      {
        if (typeof(nombre)=='undefined')
          document.write("Bienvenido al mundo cibernético");
        else
          document.write("Bienvenido de nuevo, "+nombre);
      }
    -->
  </script>
</head>
<body>
  <script language="javascript">
    <!--
document.write('<h3 align="center">MUNDO CIBERNÉTICO</h3>');
document.write('<p align="center">');
document.write('<b>Mensaje de bienvenida genérico: </b><br>');
bienvenida();
document.write('<p align="center">');
document.write('<b>Mensaje de bienvenida personalizado' +
  ' para Javier: </b><br>');
bienvenida("Javier");
    -->
  </script>
</body>
</html>
```

Ejercicio alumnado: Probar el código anterior

El alumnado hará este ejemplo en los ejercicios

[ParametrosPorDefecto.html](#)

3.7 Funciones recursivas

Javascript permite que en el cuerpo de una función se llame a la propia función, es decir, permite definir funciones recursivas. Existen diversos algoritmos cuya implementación resulta más elegante y clara si se utiliza una solución recursiva. Un ejemplo típico es el cálculo del factorial de un número.

Al aplicar la función factorial a un número N, el resultado que se obtiene es: $N * (N-1) * (N-2) * \dots * 3 * 2 * 1$. Lo que expresado de otra forma implica $N * (N-1)!$

```
function factorial(N)
{
  if (N>0)
```



```
        return N * factorial(N-1);
    else
        return 1;
    }
```

El alumnado realizará el programa factorial de un número dado en los ejercicios de la Unidad.

Ejemplo completo de manejo de funciones no ver con alumnado(lo harán):
[Funciones.htm](#) [Funciones.css](#) [Funciones.js](#)

3.8 Las funciones son datos

A partir de este apartado vamos a estudiar una serie de consideraciones especiales que se encuentra el lector cuando trabaja con jQuery, Mootools y otras librerías semejantes donde uno se encuentra con conceptos como funciones anónimas, también llamadas funciones lambda, funciones autoejecutables y otras lindeces que le hacen a uno dudar de si sabe algo de javascript. Algunos de estos conceptos volveremos a verlos en el apartado dedicado a la ¿¿programación orientada a objetos?? (clases/objetos).

En javascript, como veremos a partir de ahora, casi todo se realiza con una sola palabra clave function.

Las funciones son funciones, pero también datos: el hecho de definir una función con su propio nombre (en este caso "saluda" es el nombre de la función) hace que se cree una variable global llamada "saluda" que podemos leer y pasar como parámetro, por ejemplo:

```
function saluda() {
    alert("hola");
}

function ejecuta(func) {
    func();
}

ejecuta(saluda);
// Ojo sin comillas, estamos haciendo referencia a la función saluda
```

También podemos crear funciones como expresiones de esta manera:

```
var saluda = function(quien) {
    alert("hola "+quien);
}

saluda("mundo");
```

Con esta sintaxis lo que estamos haciendo en realidad es crear una función anónima y asignarle un nombre inmediatamente a través de una variable, por lo que es equivalente a definir la función como hacíamos al principio con function saluda (quien).

Para acabar, podemos crear funciones como expresiones y con nombre a la vez, como las funciones como declaración. La utilidad de esto es hacer funciones recursivas:

```
var f = function fact(x) {  
    if (x <= 1) return 1  
    else return x*fact(x-1);  
}  
  
f(5);
```

El alumnado realizará el programa factorial de un número dado en los ejercicios de la Unidad.

3.9 Funciones anónimas, autoejecutables y que devuelven funciones

Una función anónima se puede definir sin que sea asignada a ninguna variable:

```
function(quien) {  
    alert("hola "+quien);  
}
```

Sin embargo, hacer esto es completamente inútil: definir una función sin nombre hace que sea imposible ser ejecutada más tarde, pues sin un nombre con el que acceder a ella es imposible encontrarla.

Pero podemos ejecutarla en el mismo momento en el que la definimos. Para ello, solo tenemos que encerrar entre paréntesis, y después usar unos nuevos paréntesis con los parámetros, como hacemos con una función normal.

```
(function() { alert("hola mundo") })();
```

Por supuesto, podemos pasarle parámetros a nuestra función autoejecutable. En el siguiente ejemplo, se pasa como parámetro “mundo” a la función:

```
(function(quien) {  
    alert("hola "+quien);  
}) ("mundo");
```

Por supuesto, una función puede devolver una función anónima. Será responsabilidad del programador asignarla a una variable:

```
function saludador(quien) {  
    return function() {  
        alert("hola "+quien);  
    }  
}  
var saluda = saludador("mundo");  
saluda();
```

O podemos ejecutar la función que se ha retornado directamente, sin asignarla a ninguna variable:

```
saludador("mundo")();
```

3.10 Funciones internas o definidas dentro de otra función

Las funciones además se pueden anidar:

```
function saluda(quien) {
    function alertasaludo(quien) {
        alert("hola "+quien);
    }
    alertasaludo(quien);
}

saluda("mundo");
```

Las **funciones anidadas** se llaman **inner-private function**. Inner porque son internas, y private porque son solo accesibles desde el código de la función desde donde son definidas. En nuestro ejemplo, alertasaludo() solo se puede invocar desde dentro de saluda().

Podemos combinar funciones anidadas con funciones que retornan funciones:

```
function saludator(quien) {
    function alertasaludo() {
        alert("hola "+quien);
    }
    return alertasaludo;
}

var saluda = saludator("mundo");
saluda();
```

Y lo anterior lo podemos combinar con una función anónima auto-ejecutable.

```
var saluda = (function(quien) {
    function alertasaludo() {
        alert("hola "+quien)
    }
    return alertasaludo
})( "mundo");

saluda();
```

Todas las técnicas anteriores se pueden usar para hacer código más conciso y seguro, para prevenir conflictos con el ámbito de las variables, en programación dinámica de clases/objetos, etc.

El siguiente código es un claro ejemplo de código más seguro con funciones anónimas

```
var valores = [5, 2, 11, -7, 1];

function combineTodo(lista, valorInicial, operacion) {
    var resultado = valorInicial;
    for (var i=0; i< lista.length; i++) {
        resultado = operacion(resultado , lista[i]);
    }
    return resultado ;
}

var suma = combineTodo(valores, 0, function (a, b) {
```

```

        return a+b;
    });

    document.write(suma);
    document.write("<br />-----<br />");

    var producto = combineTodo(valores, 1, function (a, b) {
        return a*b;
    });

    document.write(producto);
    document.write("<br />-----<br />");

```

Ejemplo anterior de manejo de funciones anónimas:

[combinarOperaciones.html](#)

jQuery utiliza funciones anónimas autoejecutables para introducir varios scripts en la misma página y evitar conflictos con las variables que utilizan el mismo nombre en todos los script.

```

(function () {
    //Ponga el código del script aquí
})();

```

Eso ocurre con toda la librería jQuery, al completo, que se engloba en este código:

```

(function(window, undefined) {
    // Use the correct document accordingly with window argument (sandbox)
    var document = window.document;
    var jQuery = (function() {

        // Aquí va todo el código de jQuery

    });

})(window);

```

IMPORTANTE VER LOS ARCHIVOS: ACTUALIZACIÓN ECMASCRIPT 6

Carpeta → **Funciones flecha_ arrow function**

Repaso definición de Callbacks