# Tema 10

## setInterval, setTimeout, callback, closure

## setInterval(funcion,milisegundo[,param1,param2,.....])

Es un método del objeto window de javascript que nos permite llamar a una función cada xxx milesegundos. Esta llamada se repetirá hasta que cancelemos dicha llamada.

```
respu = setInterval(mensaje,3000);
```

Llama a la función “mensaje” cada 3000 milisegundo ( 3 segundos ). Esta función devuelve un identificador que utilizaremos para detener las llamadas.

```
<script>
window.addEventListener("load",function(){
   document.addEventListener("click",function(){
      window.clearInterval(respu);   // detener el setInterval
      alert("parado el setInterval");
   });
/// Inicio de código
   var respu;
   var a=4;
   var b=5;
   alert("hemos empezado");
   respu=setInterval(mensaje,3000,14,16);
});

function mensaje(a,b){
   console.log("mensaje " +a+b);
}
```

En este ejemplo vemos también como podemos pasarle parámetros a la función.

La definición de la función la realizamos con los parámetros que deseamos.

Cuando creamos el intervalo después del parámetro de “milisegundos” colocamos los parámetros que queremos pasar a la función.

```
respu=setInterval(mensaje,3000,14,16);
```

Estos valores que pasamos como parámetros siempre serán los mismos.

Para cancelar las llamadas a la función utilizaremos clearInterval(id). “id” es el identificador de la llamada, es decir, es el valor que devuelve setInterval.

```
respu = setInterval(mensaje,3000);
....
....
clearInterval(respu);
```

Podemos pasar variables exactamente igual que con el método setInterval.

# setTimeout(funcion, milesegundos)

Es otro método del objeto window de javascript que ejecuta una función transcurrido xxx

milisegundos y no la vuelve a ejecutar nunca más, es decir, ejecuta la función una única vez.

Al igual que en setInterval devuelve un identificador, que también lo utilizamos para cancelar setTimeout().

```
respu = setTimeout(mensaje,3000);
```

Ejecuta la función mensaje dentro de 3 segundos.

Si deseamos anular una acción teclearemos la siguiente instrucción

*clearTimeout(respu);*

# callback

"llamada de vuelta". Es un concepto que indica que a una función le podemos pasar una función como parámetro y que la ejecute cuando considere oportuno.

```
function funcion1(){
   alert("esta es la funcion 1");
   funcion2(funcion3) ;// pasamos el nombre de la función
   alert("volvemos a la funcion1");
}
function funcion2(callback){
   alert("esta es la funcion2");
   callback();
   alert("volvemos a la funcion2");
}
function funcion3(){
   alert("esta es la funcion 3");
}
funcion1();
```

Otro forma de pasar una función es trabajando con funciones anónimas

```
function funcion1(){
   alert("esta es la funcion 1");
      funcion2(function(){alert("esta es la funcion 3");}); // pasamos la
definición de función
   alert("volvemos a la funcion1");
}
function funcion2(callback){
   alert("esta es la funcion2");
   callback();
   alert("volvmos a la funcion2");
}
funcion1();
```

También podemos pasar a la función parámetros

```
function funcion1(){
   alert("esta es la funcion 1");
   funcion2(funcion3); // Pasar función SIN paréntesis
   alert("volvemos a la funcion1");
}
function funcion2(callback){
   alert("esta es la funcion2");
   callback('valor del parametro');  // pasamos el parámetro
   alert("volvmos a la funcion2");
}
function funcion3(parametro){
```

```
    alert("esta es la funcion 3 " + parametro); // utilizamos el parámetro
}
funcion1();
```

Visto de esta manera no parece muy interesante. Hagamos ahora otro ejemplo.

```
function funcion1(a,b,callback){
    return callback(a+b,a*b);
};
funcion1(5,3,function(sum,prod){
    alert(sum+ " -- " + prod);
});
funcion1();
```

Una cosa muy importante, los callback **no** son asíncronos, es decir, ejecuta el callback y hasta que no finaliza la ejecución no continua con la siguiente instrucción. Esto lo podemos observar ejecutando los ejemplos anteriores.

Ahora estudiemos un poco el ámbito de las variables

```
function funcion1(){
   var variable1;
   variable1 = "existo1";
   alert("esta es la funcion 1");
   funcion2(funcion3)
   alert("volvemos a la funcion1");
}

function funcion2(callback){
   var variable2="existo2";
   //alert("f2 variable1 " + variable1); //variable1 is not defined
   alert("esta es la funcion2");
   callback('valor del parametro');
   alert("volvemos a la funcion2");
}

function funcion3(parametro){
   alert("esta es la funcion 3 " + parametro);
   //alert("f3 variable1 "+ variable1);  //variable1 is not defined
   //alert("f3 variable2 "+ variable2);  //variable2 is not defined
}
funcion1();
```

Veamos otro ejemplo

```
function funcion1(){
   var variable1="exito";
   alert("esta es la funcion 1");
   funcion2(function(){
       salidavar2=typeof variable2 === "undefined"? "indefinida": variable2;
       alert("esta es la funcion 3 " + variable1 + " -- " + salidavar2);
   });// variable2 indefinida

   alert("volvemos a la funcion1");
}

function funcion2(callback){
   var variable2="exito2";
   alert("esta es la funcion2");
   callback();
   alert("volvemos a la funcion2");
```

```
}
funcion1();
```

En estos ejemplo queda claro que la variable es visible entre las llaves {} donde se define la variable. En los siguientes enlaces podemos ver la diferencia entre null y undefined y estudiar como trabajar con estos conceptos.

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/undefined

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/null

El callback visto con estos ejemplos parece poco útil. Se puede realizar lo mismo llamando nosotros a las funciones en el orden adecuado. En ocasiones nos interesa que una función controle cuando otra función se debe ejecutar.

Ver un ejemplo en esta página http://www.abcdisegno.com/funciones-callback/

# closures

Un closure es un tipo especial de objeto que combina dos cosas: una función, y el entorno en que se creó esa función. Los closures son funciones que manejan variables independientes. En otras palabras, la función definida en el closure "recuerda" el entorno en el que se ha creado. Explicación obtenida de

https://developer.mozilla.org/es/docs/JavaScript/Guide/Closures .

Para entender esto hagamos dos ejemplos

```
function externa(x){
   var variable=3;
   function interna(y){
      return(x+y+(++variable));
   }
   return interna(10);
}

alert(externa(2)); //16
```

Este código siempre devuelve 16. Si volvemos a ejecutar alert(externa(2)); nos vuelve a retornar 16. El código se ejecuta creando cada vez la pila de funciones de ejecución.

```
function fuera(x){
   var variable=3;
   return function(y){
      return(x+y+(++variable));
   }
}

var clausura=fuera(2);

alert(clausura(10));
```

La primera vez que ejecutemos el código nos dará 16. Si volvemos a ejecutar alert(clausura(10)); nos devolverá el valor 17 y la siguiente vez 18..... y así sucesivamente. Qué ha sucedido?. clausura es un objeto que contiene una función y mantiene el entorno de variables.

Veamos que contiene fuera y clausura:

```
//fuera:

function fuera(x){
   var variable=3;
   return function(y){
      return(x+y+(++variable));
   }
}

//closure:

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

Como vemos fuera se ejecuta entera cada vez, por tanto, variable se reinicia cada vez que llamamos a la función.

En clausura es una función anónima que, aunque no se vea, mantiene el alcance de todas las variables, por tanto, variable se inicializó y ahora vamos incrementado su valor.

Un uso que podemos dar a los closures es crear variables “privadas” en javascript.

Veamos otro ejemplo, ahora cogemos el ejemplo que podemos encontrar en w3school http://www.w3schools.com/js/js_function_closures.asp

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();
```

En este ejemplo vemos una función anónima que se ejecuta y se asigna a add. El valor que devuelve es una función que mantiene los valores que cogió la variable cuando se ejecutó por primera vez.