

Lección 20

FUNCIONES II

Function Statement



Function statement

Function statement define la función usando una declaración y sus parámetros, declarándose directamente en una sentencia. Tienen la estructura conocida:

```
function nombre_de_la_funcion ([parametros]) {  
    //codigo  
}
```

Usa la palabra reservada function, además de el nombre de la función y por último puede o no tener parámetros.

Function statement

Por ejemplo, la función `cuadrado(lado)` y la función `imprimirSaludo()`:

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

```
function imprimirSaludo() {  
    console.log()  
}
```

Function Expression



Function Expression

Otra forma de declarar funciones es a través de una expresión. Su estructura es :

```
var nombre = function ([parámetros]) {  
    //código  
}
```

Tiene la palabra reservada function, puede tener o no el nombre de la función y por último puede o no tener parámetros.

Function Expression

Por ejemplo, la función cuadrado y saludo:

```
var cuadrado = function sq(lado) {  
    return lado*lado;  
}
```

```
var saludo = function () {  
    console.log();  
}
```

Function Expression

Las funciones declaradas con function expression pueden:

- ❑ Ser asignadas a una variable
- ❑ Puede ser pasadas como parámetro
- ❑ Pueden ser enviadas en el return de una función

Funciones Statement vs Expression



Statement vs Expression

Hay tres diferencias principales entre *function statements* y *function expressions*

- ❑ Hoisting
- ❑ Ejecución
- ❑ Nombre

HOISTING








HOISTING

Statement vs Expression : Hoisting

Hay varias diferencias, la primera es que en el caso de function statement, se puede “usar” la función antes de su declaración. Esto es llamado **Hoisting** o elevación. Por ejemplo si tenemos en el código:

```
1 isHoisted();
2
3 function isHoisted(){
4     console.log("Elevada");
5 }
6
7 console.log("Después");
```

En consola producirá el resultado

 Elevada	 isHoisted — app.js:4
 Después	 global code — app.js:7
	

Statement vs Expression : Hoisting

Puedes ver el código funcionando en el siguiente gif.

```
1 isHoisted();  
2  
3 function isHoisted(){  
4     console.log("Elevada");  
5 }  
6  
7 console.log("Después");  
8 isHoisted();  
9  
10
```



Statement vs Expression : Hoisting

Si definimos la función a través de una expresión, tendremos el código:

```
1 notHoisted();  
2  
3 var notHoisted = function(){  
4     console.log("Elevada");  
5 }  
6 console.log("Después");  
7 notHoisted();
```

Lo que en consola, mostrará es un error similar, pues para la línea 1 la función no se ha definido ni elevado.

```
❗ ▶ TypeError: global code — app.js:1  
notHoisted is not  
a function. (In 'notHoisted()', 'notHoisted' is  
undefined)
```

Statement vs Expression : Hoisting

Puedes ver el código funcionando en el siguiente gif.

```
1 notHoisted();  
2  
3 var notHoisted = function(){  
4     console.log("Not Hoisted");  
5 };  
6 console.log("Después");  
7 notHoisted();  
8  
9  
10
```



Statement vs Expression : Hoisting

Si en el código anterior se comenta la primera línea que nos causaba error, el código quedaría como sigue:

```
1 // notHoisted();
2
3 var notHoisted = function(){
4   console.log("Not Hoisted");
5 };
6 console.log("Después");
7 notHoisted();
8
```

Lo que en consola mostrará lo siguiente, donde no tenemos errores, y la función ya ha sido definida y es usado luego de su definición.

Después	global code — app.js:6
Not Hoisted	notHoisted — app.js:4

Statement vs Expression : Hoisting

Puedes ver el código funcionando en el siguiente gif.

```
1 notHoisted();
2
3 var notHoisted = function(){
4     console.log("Not Hoisted");
5 };
6 console.log("Después");
7 notHoisted();
8
9
10
```

! ▶ TypeError: S global code — app.js:1
notHoisted is not a
function. (In 'notHoisted()', 'notHoisted' is
undefined)

>

EJECUCION



Statement vs Expression : Ejecución

Como saben, JavaScript es un lenguaje interpretado. La interpretación funciona en dos fases:

- ❑ Fase de compilado: el intérprete lee el código JavaScript y lo transforma a bytes o a binarios. Lo que se denomina parsing.
- ❑ Fase de ejecución: el código, que ha pasado por el parseo, es interpretado, ejecutado.

Statement vs Expression : Ejecución

Las funciones declaradas con function expression pueden:

- ❑ Ser asignadas a una variable
- ❑ Puede ser pasadas como parámetro
- ❑ Pueden ser enviadas en el return de una función

Todo esto es posible gracias que las funciones declaradas con function expression se invocan y generan en tiempo de ejecución. Es decir el browser las lee cuando son requeridas por el flujo.

Statement vs Expression : Runtime

A diferencia de las function expression, las function statement son cargadas prioritariamente en la fase de compilado del intérprete. A continuación puedes ver cómo al iniciar, las function statement ya han cargado y están listas para usar antes de ejecutar las siguientes sentencias.

En cambio la function expression se carga a penas se asigna o declara.

Puedes ver a más detalle la ejecución del gif en el [link](#).

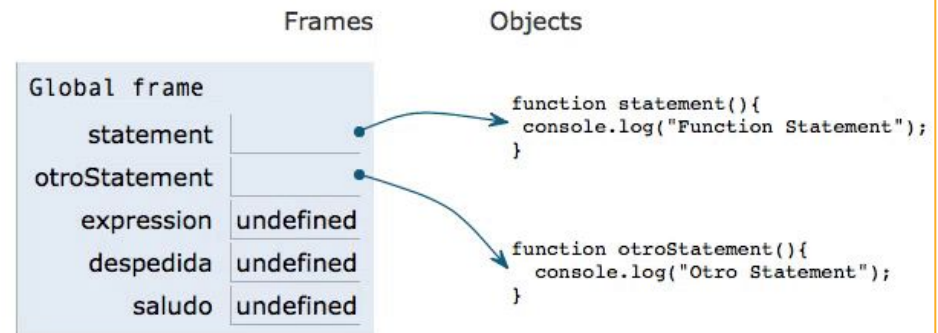
Statement vs Expression : Runtime

Al iniciar el gif, la fase de compilado se ha ejecutado, por lo que se ve en Frames, las variables definidas, así como las funciones statement.

JavaScript

```
1 // Function Statement
2 statement();
3
4 function statement(){
5   console.log("Function Statement");
6 }
7
8 function otroStatement(){
9   console.log("Otro Statement");
10 }
11 //Function Expression
12 var expression = function (){
13   console.log("Function Expression");
14 }
15
16 expression();
17 var despedida;
18 var saludo = "Hola";
```

Print output (drag lower right corner to resize)



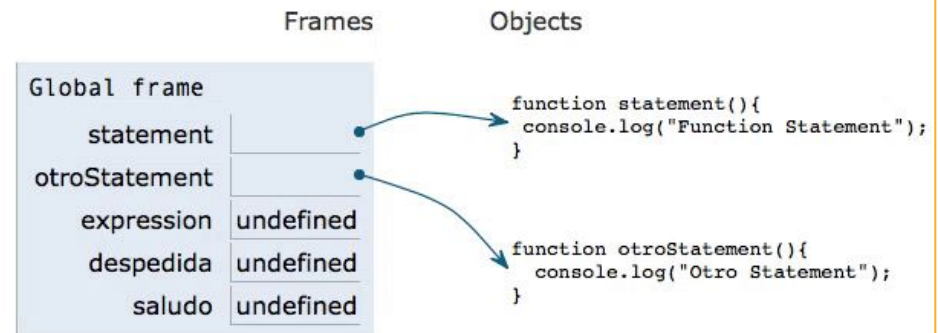
Statement vs Expression : Runtime

Cuando llega a la línea 12 de la ejecución, se puede notar cómo es que se crea la función en el caso de la función expression, esto ya es en fase de ejecución.

JavaScript

```
1 // Function Statement
2 statement();
3
4 function statement(){
5   console.log("Function Statement");
6 }
7
8 function otroStatement(){
9   console.log("Otro Statement");
10 }
11 //Function Expression
12 var expression = function (){
13   console.log("Function Expression");
14 }
15
16 expression();
17 var despedida;
18 var saludo = "Hola";
```

Print output (drag lower right corner to resize)



NAMING



Naming

A diferencia de la function expression, para function statement se requiere que la función tenga un nombre.

En el caso de function expression, el nombre de la función es opcional.

```
var cuadrado = function (lado) {  
    return lado*lado  
}
```

Funciones Anónimas



Funciones Anónimas

Los function expression no requieren que la función tenga nombre, por ejemplo:

```
var cuadrado = function() {  
    return lado*lado;  
}
```

Y si además la función no entrega data significativa se tiene funciones como la que sigue:

```
function() {  
    // code  
}
```

Callbacks



Callbacks

Las funciones Callback, son funciones que se pasan a otras funciones como un parámetro, para luego ser ejecutadas.

```
var friends = ["Giuli", "Cesar", "Diego",  
"Edward"];  
friends.forEach(function (eachName, index){  
    console.log(index + 1 + ". " + each Name);  
    // 1. Giuli, 2. Cesar, 3. Diego, 4. Edward
```

Por ejemplo, en el for each, se está enviando una función anónima como parámetro. Esta función imprimirá los nombres del array.

Callbacks: Recomendaciones

Al usar callbacks debes recordar:

- ❑ Son funciones que pueden ser anónimas o tener un nombre y ser referenciadas.
- ❑ Antes de ejecutarla, debes asegurarte que lo que se te ha pasado en el parámetro es una función.

```
function getInput(options, callback) {  
  allUserData.push(options);  
  // Asegurate de revisar si callback es una función  
  if (typeof callback === "function") {  
    // Llama a la función ahora que sabes que es una  
    función.  
    callback(options);  
  }  
}
```

- 

```
function nombre_de_la_funcion.call (objeto_this,  
                                     parametros)
```

- 

```
function nombre_de_la_funcion.apply (objeto_this, [array  
con parámetros])
```

Callbacks: Recomendaciones

Usando apply

```
// Nota que tenemos un parámetro extra callbackObj
function getUserInput(nombre, apellido, callback,
callbackObj) {
    // La función apply hará de callbackObj sea
    // this dentro del ámbito de la función callback
    callback.apply (callbackObj, [firstName,
    lastName]);
}
```


Closures



Closures

En JavaScript puedes escribir funciones dentro de una función, o anidar.

Un closure es una función que tiene acceso al ámbito de su función padre, incluso después de que la función padre terminó de ejecutarse.

Closures

```
function padreFuncion() {  
    var name = "Mozilla";  
    function closureFuncion() {  
        alert(name);  
    }  
    return closureFuncion;  
}  
  
var final = padreFuncion();  
final();
```

La función padreFuncion, retorna una función (closureFuncion), no la ejecuta.

Asignamos el resultado de padreFuncion, la función closureFuncion, a final.

Cuando ejecutamos final, esta nos devolverá un alert con la palabra "Mozilla".

Frames y Closures

En la fase de ejecución, el intérprete de JavaScript genera espacio lógicos para definir el concepto de ámbitos. A esto le llamamos Frames.

Los closures acceden a sus propios elementos, y hacen referencia a elementos superiores de los Frame en los que están contenidos.

Frames y Closures

En naranja, tenemos el Frame Global.

En azul, tenemos el Frame de primeraFuncion.

En verde, en Frame que genera la función imprimir.

Por último, en rosado, el Frame que genera la función nuevaImpresion.

Puedes ver cómo se generan los Frames en el [link](#)

```
function primeraFuncion(iteraciones){  
  var message = "Primer mensaje";  
  var imprimir = function (veces){  
    console.log("Iteraciones :" + veces );  
    console.log(message);  
  }  
  
  function nuevaImpresion(){  
    console.log("Nueva Impresión");  
  }  
  
  for (var i=0; i<iteraciones; i++){  
    imprimir(i);  
    nuevaImpresion();  
  }  
}  
  
primeraFuncion(1);
```

Closures

JavaScript

```

1 function primeraFuncion(iteraciones){
2
3   var message = "Primer mensaje";
4
5   var imprimir = function (veces){
6     console.log("Iteraciones :" + veces );
7     console.log(message);
8   }
9
10  function nuevaImpresion(){
11    console.log("Nueva Impresión");
12  }
13
14  for (var i=0; i<iteraciones; i++){
15    imprimir(i);
16    nuevaImpresion();
17  }
18 }
19
20 → primeraFuncion(1);

```

[Edit code](#) | [Live programming](#)

line that has just executed

next line to execute

! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to go there.

<< First < Back **Step 1 of 15** Forward > Last >>

Print output (drag lower right corner to resize)

Frames

Objects

Global frame
primeraFuncion

```

function primeraFuncion(iteraciones){
  var message = "Primer mensaje";

  var imprimir = function (veces){
    console.log("Iteraciones :" + veces );
    console.log(message);
  }

  function nuevaImpresion(){
    console.log("Nueva Impresión");
  }

  for (var i=0; i<iteraciones; i++){
    imprimir(i);
    nuevaImpresion();
  }
}

```

Closures

Podemos notar que la función imprimir es un closure. Esto debido a que usa una variable que no está en su ámbito o frame (message), pero que sí existe en el ámbito de primeraFuncion.

```
function primeraFuncion(iteraciones){  
    var message = "Primer mensaje";  
    var imprimir = function (veces){  
        console.log("Iteraciones :" + veces );  
        console.log(message);  
    }  
    function nuevaImpresion(){  
        console.log("Nueva Impresión");  
    }  
    for (var i=0; i<iteraciones; i++){  
        imprimir(i);  
        nuevaImpresion();  
    }  
}  
  
primeraFuncion(1);
```

Closures

Los closures son algunos de los conceptos más usados en JavaScript, ya que los closures se crean cuando se llama a la función que los acoge. Esto permite tener flexibilidad, objetos más independientes, manejo de estados y modularizar. Pero también puede generar que se use demasiada memoria, debido a que todas las variables y manejos se generan nuevamente apenas se use la función.