UD1. Javascript

FUNCIONES

Funciones

Declaración de una función
(En tiempo de compilación)

```
function suma_y_muestra(numero1,numero2) {
 let resultado = numero1 + numero2;
 alert("El resultado es: " + resultado);
 return resultado;
}
```

- JS **no** da error si llamas a una función con **más argumentos**.
- El orden de los argumentos es **importante**.
- Las funciones pueden utilizar y modificar variables globals. (side-effects)
- Puede o no tener un **return**.
- Con **()** invocas a la función, sin ella al objeto que representa.
- Una función es un **objecto**.

Funciones como variables

```
function toCelsius(fahrenheit) { return (5/9) * (fahrenheit-32); }
var x = toCelsius(77);
console.log(`La temperatura es: ${x} C`);

// Directamente en la variable
x = function toCelsius(fahrenheit) { return (5/9) * (fahrenheit-32); }
console.log(`La temperatura es: ${x(77)} C`);

// Sin el nomnbre de la función
x = function (fahrenheit) { return (5/9) * (fahrenheit-32); }
console.log(`La temperatura es: ${x(77)} C`);
```

Ámbito de las funciones

- Las funciones tienen que estar en **el ámbito en el que son llamadas**.
- Las funciones pueden ser declaradas **después de definirlas** (con la sintaxi de la declaración de función).
- Les funciones **no** pueden ser declaradas **después** si se definen con una expressión de función.

```
console.log(square(5)); // 25
/* ... */
function square(n) { return n*n }
```

```
console.log(square); // undefined
console.log(square(5)); // TypeError
let square = function (n) {
 return n * n;
}
```

Ámbito (Scope)

Local o de funció

```
function local() {
 var a = 2;
 console.log(a);
}
local();
console.log(a);
```

Global

```
var a = 1;
function global() {
 console.log(a);
}
global();
console.log(a);
```

De bloc

```
for (let i = 0; i < 10; i++) {
 console.log(i);
}
console.log(i); // error
```

Ámbito de las variables en funciones

- Una variable en una función **no** puede ser accedida desde **otro lugar**.
- Una función puede **acceder** a las **variables globales** o a las de una **función padre**.
- Las funciones pueden ser **anidadas**, esta será la manera de hacer **variables privadas**.

```
function addSquares(a,b) {
 function square(x) {
   return x * x;
 }
 return square(a) + square(b);
}
a = addSquares(2,3); // decuelve 13
b = addSquares(3,4); // devuelve 25
c = addSquares(4,5); // devuelve 41
```

```
function outside(x) {
   function inside(y) {
     return x + y;
   }
   return inside;
  }
let fn_inside = outside(3);
let result = fn_inside(5); // devuelve 8
let result1 = outside(3)(5); // devuelve 8
```

Hoisting

- Permite utilitzar variables o funciones antes de ser declaradas.
- Los lenguajes compilados tienen hoising y los interpretados no, no obstante **JS** es un **híbrido** que tiene un “precompilado” y por tanto, tiene hoising.
- En **JS** se permite con ***“var”*** y con ***“funciones”***. Con ***“let”*** y ***“const”*** **no** se permite utilitzarlo antes.
- En cualquier caso, es mejor siempre **respetar el orden** declarando variables globales, funciones globales… antes del código que se ejecutará.

https://www.escuelafrontend.com/articulos/hoisting-ejemplos-practicos

Funciones anónimas

- Cuando **no** necessitas que la función sea llamada desde otro lugar.
- Para pasar una función como argumento de otra función.
- Para guardar una función en una variable (no Hoising)

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
var foo = function(){ /*...*/ };
```

https://es.acervolima.com/funciones-anonimas-de-javascript/

Constructor Function

```
var suma = new Function('a','b',"return a + b ");
console.log(suma(10,20));
```

- **No recomendado**
- Es menos eficiente porque se crea en ejecución y no en compilación.
- Puede dar problemas de seguridad como **eval()**

Funciones flecha

- Simplificación de las funciones anónimas.
- No se necesita escribir **function, return ni {}**
- No se comportan como objectos ni tienen **this.**
- Se recomienda utilizar ***"const"***, ya que siempre son constantes.
- Si tienen más de una instrucción se necesitan los **{}** y el **return**.
- **No** se pueden hacer métodos (al no tener this, no pueden acceder al objeto) .

```
// ES5
var x = function(x, y) {
 return x * y;
}
// ES6
const x = (x, y) => x * y;
```

```
persona = { nombre: 'Pepe',apellido: 'Garcia',
  consulta: function () { return `${this} ${this.nombre} ${this.apellido}`},
  consultar: () => `${this} ${this.nombre} ${this.apellido}`
}
console.log(persona.consulta(), persona.consultar());
```

Funciones auto-invocadas

- Si ponemos **()** en la definición de una función, esta se crea y se ejecuta en el momento, **sin que nadie la llame**.
- Las variables **no son accesibles desde fuera de la función**.
- Acepta argumentos con los **()** del final.
- **Recomendable** para el ***"main"*** de la aplicación.

```
(function () {
 var aName = "Barry";
})();
aName // "Uncaught ReferenceError: aName is
not defined"
var result = (function () {
 var name = "Barry";
 return name;
})();
result; // "Barry"
```

Argumentos por defecto

```
var x = function(x=2, y=2) {
 return x * y;
}
var multi = function(x,y){
   if (x === undefined) {x=2;}
   if (y === undefined) {y=2;}
   console.log(arguments.length); // és un array
   return x*y;
}
```

Call, Apply, Bind

- **Call**: Para llamar a una función indicando cual es su **contexto** de ejecución.
- **Apply**: Igual que Call, pero se pueden enviar los argumentos como un array.
- **Bind**: Permite crear una función con el contexto indicado.

```
function Car(type, fuelType){
 this.type = type;
 this.fuelType = fuelType;
}
function setBrand(brand){
 Car.call(this, "convertible", "petrol");
 this.brand = brand;
 console.log(`Car details = `, this);
}
const newBrand = new setBrand('Brand1');
```

```
this.x = 9;
var module = {
 x: 81,
 getX: function() { return this.x;
}
};
module.getX(); // 81
var getX = module.getX;
getX(); // 9
var boundGetX = getX.bind(module);
boundGetX(); // 81
```