

Fundamentos de JavaScript

Sintaxis básica. Tipos de datos

Tipos primitivos. Conversión.

Operadores y expresiones. Tipos referencia

Funciones. Control de flujo

- Sintaxis básica. Tipos de datos
- Tipos primitivos. Conversión.
- Operadores y expresiones. Tipos referencia
- Funciones. Control de flujo



Sintaxis básica

jueves, 18 de mayo de 2017 11:39

Sintaxis.

No se tienen en cuenta los espacios en blanco y las nuevas líneas

Sensible al caso (distingue las mayúsculas y minúsculas),
e.g. en los nombres de variables, funciones y operadores

No es necesario terminar cada sentencia con el carácter de **punto y coma**,
aunque es habitual y se recomendaba hacerlo hasta ES6;

Documentación del software

Como en cualquier lenguaje de programación, se define un carácter o
caracteres que permiten introducir comentarios

// Línea de comentario

/* Varias líneas
de comentario */

Palabras reservadas

Js

abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while

Sorprendentemente,
no lo son
undefined,
NaN,
Infinity.

Modo estricto ("use strict")

El modo estricto de ES 5 es una forma de optar explícitamente por una variante restringida de JavaScript, con una semántica intencionalmente diferente que el código normal.

- elimina algunos errores silenciosos de JavaScript haciendo que lancen excepciones.
- corrige errores que hacen que sea difícil para los motores de JavaScript realizar optimizaciones: a veces, el código escrito en modo estricto puede correr más rápido que el que no es estricto.
- prohíbe cierta sintaxis que es probable que sea definida en futuras versiones de ES, añadiendo las palabras reservadas implements, interface, let, package, private, protected, public, static, y yield.

Constantes y variables

domingo, 11 de marzo de 2018 22:34

En cualquier lenguaje de programación existen **contenedores de datos**

- Constantes : no pueden variar a lo largo del programa
- Variables : si pueden hacerlo

Los atributos de un objeto son un caso particular de este tipo de contenedores

La mayoría de los lenguajes distingue

- Declaración : se establece el nombre (y el tipo y tamaño de la variable)
- Inicialización : se establece el valor inicial de la variable (y su tipo si aún no estaba definido)

Una variable declarada con tipo suele tener un valor por defecto

Una constante suele declararse / inicializarse forzosamente a la vez

Sintaxis de variables y constantes

El nombre de una variable o constante también se conoce como identificador y debe cumplir las siguientes normas

Sólo puede estar formado por

- Letras
- Números
- \$ (dólar)
- _ (guion bajo).

El primer carácter no puede ser un número.

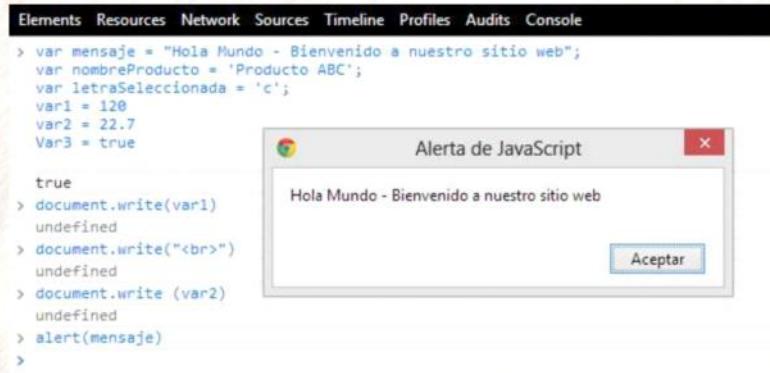
Los nombres de variables suelen utilizar **camelCase**

Los nombres de constantes suelen utilizar letras **MAYSCULAS**

1. Declaración. Inicialización o asignación
2. Alcance (scope). var
 1. Variables globales
 2. Variables locales a funciones
3. Alcance de bloque. let (ES6)
4. Constantes (Alcance de bloque) const (ES6)

Ejercicio - Consola

Variables en una consola: Abrimos la consola de un navegador (e.g. Chrome) apuntando a la dirección <about:blank> (página en blanco) y comprobamos como se pueden declarar e inicializar variables, para luego utilizarlas como parámetros de la función `alert()` o presentarlas en la parte superior de la pantalla con `document.write()`



The screenshot shows a browser's developer tools open to the 'Console' tab. The console window has a black header bar with tabs: Elements, Resources, Network, Sources, Timeline, Profiles, Audits, and Console. Below the header, there is a scrollable list of JavaScript commands and their results. The commands include declarations for variables like `mensaje`, `nombreProducto`, `letraSeleccionada`, `var1`, `var2`, and `Var3`, and calls to `document.write` and `alert`. To the right of the console, a small 'Alerta de JavaScript' (JavaScript Alert) dialog box is displayed, containing the message 'Hola Mundo - Bienvenido a nuestro sitio web' and a 'Aceptar' (Accept) button.

```
Elements Resources Network Sources Timeline Profiles Audits Console
> var mensaje = "Hola Mundo - Bienvenido a nuestro sitio web";
> var nombreProducto = 'Producto ABC';
> var letraSeleccionada = 'c';
> var1 = 120
> var2 = 22.7
> Var3 = true

true
> document.write(var1)
undefined
> document.write("<br>")
undefined
> document.write (var2)
undefined
> alert(mensaje)
>
```

Objetivo: Comprobar el uso de variables en una consola JavaScript

Declaración. Inicialización de variables

martes, 13 de marzo de 2018 19:10

Declaración

```
var / let / const nombre_de_variable  
var / let / const nombre_de_variable= valor
```

La declaración, además del nombre de la variable, puede incluir su inicialización con un valor

Inicialización

```
nombre_de_variable= valor
```

Se establece el valor inicial de la variable.

Aunque no es recomendable, es posible

```
cualquier_nombre(*) = valor
```

(*) La variable puede no haber sido declarada previamente

Alcance (scope)

martes, 13 de marzo de 2018 19:13

Variables Globales

- Definidas fuera de cualquier función
- Declaradas sin inicializar dentro de cualquier función

```
var global = 12
let global = 12
global = 12

function calcular(pData) {
  // si se inicializa una variable SIN DECLARAR
  // dentro de una función
  // se crea como variable global
  global = 12
  global = global + pData
  // global += pData
  return global;
}
```

Variables Locales

- Definidas dentro de una función
 - o Con var -> su ámbito será la función
 - o Con let / const su ámbito será el bloque en el que se crea.

var declara / inicializa
una variable cuyo ámbito
es la función en la que
se encuentra, usándose
en este caso para
declarar un array

const y let son
novedades de ES6,
siendo su ámbito el
bloque {} en el que
se declaran

línea que daría error por hacer
referencia a una variable en un ámbito
en el que no existe

// Ejemplo de código en ES6

```
function calcular() {

  var data = [{precio: 12}, {precio: 34}, {precio: 19}];

  data.forEach( elem => {
    if (true) {
      const iva = 1.16
      let precioFinal = elem.precio * iva

      console.log(`Oferta:
        El precio final es ${precioFinal}`);
    }
  })
}

// console.log (iva)
});
```

Constantes

martes, 13 de marzo de 2018 19:11

Declaración / Inicialización

```
const nombre_de_variable= valor
```

La declaración, además del nombre de la variable, tiene que incluir necesariamente su inicialización con un valor, que no podrá cambiar en ningún momento

Existe un objeto Math que incluye entre sus atributos una serie de constantes matemáticas

Constantes matemáticas

Js

Constante	Valor	Significado
Math.E	2.718281828459045	Constante de Euler, base de los logaritmos naturales y también llamado <i>número e</i>
Math.LN2	0.6931471805599453	Logaritmo natural de 2
Math.LN10	2.302585092994046	Logaritmo natural de 10
Math.LOG2E	1.4426950408889634	Logaritmo en base 2 de Math.E
Math.LOG10E	0.4342944819032518	Logaritmo en base 10 de Math.E
Math.PI	3.141592653589793	Pi, relación entre el radio de una circunferencia y su diámetro
Math.SQRT1_2	0.7071067811865476	Raíz cuadrada de 1/2
Math.SQRT2	1.4142135623730951	Raíz cuadrada de 2

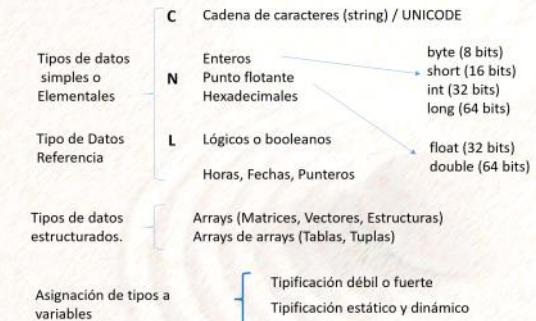
Tipos de datos

jueves, 18 de mayo de 2017 11:41

Tipos de datos

- i. Elementales y referencias. Mutabilidad
- ii. Elementales
 - 1) Number
 - 2) String
 - a) Comillas dobles y simples
 - b) Template strings (ES6)
 - 3) Boolean
 - 4) Observable (ES7)
 - 5) Undefined y null

Fundamentos. Tipos de datos



Datos en JavaScript: tipos



Asignación de tipos a variables

Tipificación **débil**: al inicializar la variable

Tipificación **dinámica**: cambia al asignar un tipo diferente

Tipo de Datos Primitivos (elementales)

- simples elementos “atómicos” de información
- se manipulan “por valor”
- son inmutables

string
number
boolean
undefined
null

Tipo de Datos Referencia

- objetos formados por múltiples elementos de información
- se manipulan por referencia
- cambian de forma dinámica

→ object

function
array
date
regexp
error...

Acceso a datos por valor

Js

A los datos primitivos se accede **por valor**

Las variables correspondientes a los tipos primitivos son accedidas por valor, es decir que **se manipula directamente el valor real almacenado en la variable**; lo único que importa es el valor en sí

Técnicamente, cada variable es un espacio de memoria diferente, donde se almacena un cierto valor, con independencia de cualquier otro

Cuando se asigna una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda.
(Sería el procedimiento supondríamos por defecto, intuitivamente)

Copia de datos por valor

Js

Al copiar los datos primitivos,
se manipulan **por valor**

```
var var_1 = 3;
```

var_1

3

```
var var_2 = var_1;  
var var_3 = var_2 + 5;
```



var_1

var_2

var_3

3
3
8

// Ahora var_3 = 8 y
var_1 sigue valiendo 3

Acceso a datos por referencia

A los tipos de referencia se accede **por referencia**

Técnicamente, el conjunto de valores que constituyen un objeto (o cualquier tipo referencia) se almacenan en un espacio de memoria al que no se puede acceder directamente, sino mediante referencias.

Al manipular un objeto, realmente trabajamos siempre con las referencias a ese objeto, más que con el propio objeto real

Cuando asignamos a un tipo de referencia otra variable del mismo tipo, la asignación se realiza por referencia: las dos variables hacen referencia (apuntan) a un mismo espacio mencionado antes.

Por lo tanto, las dos variables quedan "unidas" y hacen referencia al mismo objeto, al mismo dato de tipo referencia. Si se modifica el valor de una de ellas, el valor de la otra variable se verá automáticamente modificado

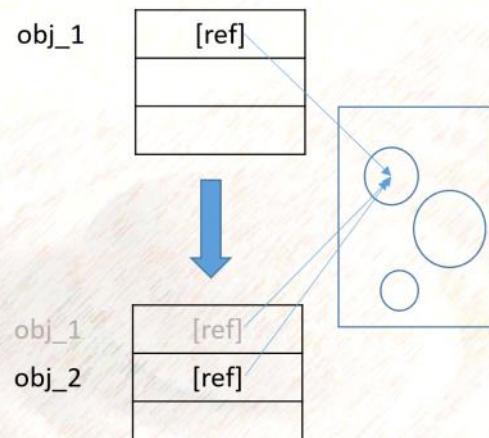
Copia de datos por referencia

los tipos de referencia se manipulan **por referencia**

```
var obj_1 = new Date(2009,11,25);
// obj_1 = 25 diciembre 2009

var obj_2 = obj_1;
// obj2 = 25 diciembre 2009

obj_2.setFullYear(2010,11,31);
// obj2 = 31 diciembre 2010
// Ahora obj1 también es 31
diciembre 2010
```



Comprobación del tipo de dato

Js

El operador **typeof**

Los posibles valores de retorno del operador son

- **undefined, boolean, number, string** para cada uno de los tipos primitivos
- **object** para los valores de referencia y también para los valores de tipo null.

```
var variable1 = 7;
typeof variable1; // "number"
var variable2 = "hola mundo";
typeof variable2; // "string"
```

Función **isNaN()**,

devuelve true si el parámetro que se le pasa no es un número

Conversión de tipos. Paso a string: String()

Js

convertir el valor de una variable de un tipo a otro, se denomina **typecasting o casting**

en JavaScript, siempre que es posible, este proceso se realiza de forma automática cuando es necesario en el curso de una operación

```
var found = true;  
alert(found)  
typeof found  
// devuelve "boolean"
```

Alerta de JavaScript

true

La salida por pantalla de alert() es un string

String(), es una función de casting “por defecto” que el sistema invoca automáticamente cuando necesita convertir cualquier valor de una variable en un *string*.

También puede utilizarse explícitamente

```
var found = true;  
var cFound = String(found);  
typeof cFound; // devuelve "string"
```

Conversión de tipos: paso a string con `toString()`

Js

`toString()` es un método, disponible en numbers, Booleans, objects, y strings que permite convertir variables de cualquiera de estos tipos en variables de cadena de texto. Es equivalente a la función de casting `String()`

```
var found = true;
alert (found.toString()); //the string “true”
var age = 11;
alert (age.toString()); //the string “11”
```

Como único argumento, en caso de numbers, acepta la base en que se encuentra el valor a convertir (por defecto 10)

```
var num = 10
alert(num.toString(16)); //”a”
```

Conversión de otros tipos a números: Number()

Js

Number(), es una función de casting “por defecto” que convierten cualquier variable en un número.

- se aplica a un **booleano** → true se devuelve como 1
false se devuelve como 0
- se aplica a un **null** → devuelve 0
- se aplica a un **undefined** → devuelve NaN
- se aplica a un **number** → devuelve el número sin modificar
- se aplica a un **string vacío ""** → devuelve 0
- se aplica a un **string** → se procesa secuencialmente,
como sigue

Conversión de otros tipos a números: Number()

Js

La conversión numérica de una cadena se realiza **carácter a carácter** empezando por el de la primera posición.

- Si el primer carácter es un +, - ó 0x se interpreta como signo o hexadecimal
- Si un carácter es un número o el signo de decimales (.), se continúa con los siguientes (los "ceros" a la izquierda se ignoran).
- Si un carácter **no es un número**, la función devuelve el valor **NaN**.

Cuando se aplica a un **objeto**, se invoca el método **valueOf()** y el valor devuelto se convierte de acuerdo con las anteriores reglas

```
var num1 = Number("Hello world!"); //NaN  
var num1 = Number("1432567K"); //NaN  
var num2 = Number(""); //0  
var num3 = Number("000011"); //11  
var num4 = Number(true); //1
```

Otra opción de conversión para strings de números es el operador unitario de signo positivo (+), que invoca la función Number()

Conversión de otros tipos a números: parse_()

Js

parseInt() y **parseFloat()**, que convierten el string que se les indica en un número entero o un número decimal respectivamente

```
var num = parseInt(22.5); //22
var num = parseInt("1234azul"); //1234
var num = parseInt("azul"); //NaN
var num = parseInt(""); //NaN
var num = parseInt("70"); //70 - decimal
var num = parseInt("0xA"); //10 - hexadecimal
var num = parseFloat(22.5); //22.5
var num = parseFloat("22.5"); //22.5
```

La conversión numérica de una cadena se realiza carácter a carácter empezando por el de la primera posición. Si el primer carácter es un número, + o -, se continúa con los siguientes. Se ignoran los espacios y si otro carácter no es un número, la función devuelve el valor obtenido hasta ese momento o NaN si se trata del primer carácter. Un string vacío también devuelve NaN.

Tipos referencia

jueves, 18 de mayo de 2017 11:44

- Tipos referenciados. Objetos
- 1) Object. Literales JSON
 - 2) Arrays
 - 3) Set y Map (ES6)
 - 4) Objetos Date, RegExp y Error
 - 5) Objetos "estáticos" Math y JSON
 - 6) Funciones

Datos de tipo referencia

Js

Todos los datos que no se ajustan a los tipos simples son considerados de **tipo referencia** (i.e. clases)

En OOP, los objetos son **instancias de la clase** a la que pertenecen.

Sin embargo, en JavaScript **no se define el concepto de clase**

Los tipos de referencia se asemejan a las clases de otros lenguajes de programación y es frecuente referirse a ellos como clases

Objeto

Una colección de elementos (propiedades), cada una con un nombre y un valor

Dato primitivo (número, cadena o booleano)

Objeto

Técnicamente es una referencia al área de memoria donde se almacenan el conjunto de elementos que lo componen.

Tipos Referencia (Clases) predefinidos

Js

String	Objetos genéricos correspondientes a los tipos primitivos
Number	
Boolean	
Object	Objeto genérico
Function	
Method	Objetos que tiene código ejecutable asociado a ellos
Constructor	
Array	Objetos en los que los elementos están ordenados numéricamente (desde el 0)
Date	Objetos que representan fechas y horas
RegExp	Objetos que representan expresiones regulares
Error	Objetos que representan errores de sintaxis o en tiempo de ejecución
Math	
JSON	"Clases" no instanciables

Arrays

Js

Los arrays son un tipo de objeto, en el que la agrupación de diversas variables (propiedades) tiene un carácter secuencial
En este caso cada propiedad esta asociada a una posición numérica dentro del array, siempre **contando desde 0**.
Finalmente, los arrays **no incorporan métodos**.

Como objetos literales, utilizando la notación JSON, mediante el operador **[]**, que delimita conjunto de valores o elementos que constituyen el array

```
var array1 = [valor0, valor1, ..., valorN];
```

```
var aArray1 = [2, "hola", true, 45.34];
```

En JSON es frecuente prescindir de la declaración.

Operadores y expresiones

jueves, 18 de mayo de 2017 11:43

- Operadores y expresiones
 - i. Aritméticos
 - ii. Relacionales
 - iii. Lógicos o de combinación
 - iv. De asignación
 - v. De concatenación
 - vi. Operador ternario o condicional
 - vii. Operador de tipo de datos `typeof`

Fundamentos. Instrucciones.

Instrucciones (Statement)

Simples

- declaración de variables
- asignación de valores: `assignment: A := A + 5`
- `assertion: assert(ptr != NULL);`
- llamada a funciones y métodos: `call...`
- devolución de valores y del control: `return ...`

Compuestas (bloques de instrucciones)

- creación de bloques de código: `block: begin ... end`
- creación de bifurcaciones en función de una condición determinada: `if / switch`
- recorrido en bucle de bloques de código: `while / do / for:`



Las instrucciones (statements) no devuelven resultados, sino que son ejecutadas únicamente por los efectos que tienen (side effects).

Las expresiones (expressions) siempre devuelven un resultado y a menudo no tienen efectos (side effects) en si mismas.

Fundamentos. Expresiones



Las variables, constantes (y literales) junto con los operadores se agrupan en **expresiones**, que se evalúan y resultan en un dato de un tipo determinado

- Números
- Cadenas de caracteres (String)
- Valores lógicos (al evaluar expresiones de comparación)

Un caso particular:
Operadores de cadena

Comparación de cadenas
(string comparison)

Comparison	Syntax
Contains only	CO
Contains any	CA
Contains string	CS
Contains pattern	CP
Contains not only	CN
Contains not any	NA
Contains no string	NS
Contains no pattern	NP

Fundamentos. Operadores básicos.



Operadores Aritméticos

suma	resta
producto	división
módulo (Resto)	
incremento	decremento

Asignación (e.g. =)

Asignación y operación aritmética
(e.g. += -= *= /= %=)

Operadores de asignación

Operadores Relacionales

Is equal to	==
Is not equal to	!=
Is greater than	>
Is less than	<
Is greater than or equal to	>=
Is less than or equal to	<=

AND
OR
NOT

Operadores Lógicos

Operadores de asignación, incremento y decremento

Js

Asignación: → **var1 = [valor]**

```
var var1 = 14
```

Incremento

++var1

var1++

Decremento

--var1

var1--

Como prefijo, el valor se incrementa / decremente antes de realizar cualquier otra operación

```
var var1= 5;  
++var1; // var1= 6  
var1--; // var1= 5
```

Como sufijo, el valor se incrementa / decremente después de realizar todas las operaciones en la sentencia

```
var var1 = 5;  
var var2 = 2;  
Var var3 = var1++ + var2;  
// var3 = 7, var1 = 6
```

```
var var1 = 5;  
var var2 = 2;  
var var3 = ++var1 + var2;  
// var3 = 8, var1 = 6
```

Operadores aritméticos

Js

+ → suma

- → resta

***** → producto

/ → división

% → módulo (Resto)

```
var var1 = 5;  
var1 = var1 + 3 // var1 = 8  
var1 = var1 - 1 // var1 = 4  
var1 = var1 * 2 // var1 = 10  
var1 = var1 / 5 // var1 = 1  
var1 = var1 % 4 // var1 = 1
```

Asignación con operación

notación abreviaba por combinación de los operadores aritméticos y de asignación

```
var var1 = 5;  
var1 += 3; // var1 = var1 + 3 = 8  
var1 -= 1; // var1 = var1 - 1 = 4  
var1 *= 2; // var1 = var1 * 2 = 10  
var1 /= 5; // var1 = var1 / 5 = 1  
var1 %= 4; // var1 = var1 % 4 = 1
```

Sobrecarga del operador +

- Suma de enteros (operador binario)
- Signo de un número (operador unitario)
- Concatenación de strings (operador binario)

Unido a la débil /
dinámica tipificación,
origina algunas
particularidades,
relacionadas con la
conversión de tipos

$13 + 7 \Rightarrow 20$
 $"13" + "7" \Rightarrow "137"$
 $"13" + 7 \Rightarrow "137"$
 $+ "13" + 7 \Rightarrow 20$

Ejercicio (3a / Consola)

Operadores aritméticos: Siguiendo el modelo del ejercicio 2, Crear un script que realice las siguientes operaciones y las muestre por pantalla

Operaciones aritméticas

- Sumar: $5 + 6 = 11$
- Restar: $3 - 8 = -5$
- Multiplicar: $5 * 6 = 30$
- Dividir: $25 / 2 = 12.5$
- Resto de la división: $25 / 2 = 1$
- Incrementar Número=5 : $\text{Número}++ = 6$
- Incrementar Número=6 : $\text{++Número} = 7$
- Decrementar Número=7 : $\text{--Número} = 6$

Alternativa. Comprobar el uso de estos operadores mediante la **consola** de JavaScript

Objetivo: Familiarizándonos con el uso de los operadores aritméticos. Comprobar como el mismo script puede ejecutarse por consola.

Operadores relacionales

JS

== → es igual a

Evalúan la comparación entre dos términos para dar como resultado un valor lógico: true o false

!= → no es igual a

> → es mayor que

< → es menor que

>= → es mayor o igual que

<= → es menor o igual que

== → es idéntico a (igual sin necesidad de conversión)

!= → no es idéntico a

? → operador ternario condicional

`(expr1) ? (expr2) : (expr3);`

Devuelve expr2 si expr1 y expr3 si !expr1

Ejercicio (3b / Consola)

Operadores relacionales:

Siguiendo el modelo del ejercicio 3, crear un script que realícelas siguientes operaciones y las muestre por pantalla

Operadores relacionales

```
var Edad = 32
```

- Igualdad: **Edad == 33** : false
- Distinto: **Edad != 33** : true
- Mayor que: **Edad > 25** : true
- Mayor o igual que: **Edad >= 32** : true
- Menor que: **Edad < 45** : true
- Menor o igual que: **Edad <= 45** : true

Alternativa. Comprobar el uso de estos operadores mediante la **consola** de JavaScript

Objetivo: Familiarizándonos con el uso de los operadores relacionales. Comprobar como el mismo script puede ejecutarse por consola.

Operadores lógicos

Js

&& → AND

|| → OR

! → NOT

var1	var2	var1 && var2	var1 var2
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

variable	!variable
true	false
false	true

Aplicación a números y cadenas

**Operadores
relacionales y lógicos:**
Siguiendo el modelo del ejercicio 3, Crear un script que realícelas siguientes operaciones y las muestre por pantalla

Operadores lógicos

AND && OR || NOT !

var Stock=2500; --- var Precio=26;

- Precio > 20 && Precio < 30 : true
- Precio > 20 && Precio < 25 : false
- Stock == 2000 || Stock == 2500 : true
- Stock != 2000 || Stock != 2500 : true
- (Stock >= 2000 && Stock <= 2200) || Precio > 25 : true

Alternativa. Comprobar el uso de estos operadores mediante la **consola** de JavaScript

Objetivo: Familiarizándonos con el uso de los operadores relacionales y lógicos. Comprobar como el mismo script puede ejecutarse por consola.

Operadores de bits

Js

a & b → **Y**: Se ponen a 1 los bits que están a 1 en \$a y \$b.

a | b → **O**: Se ponen a 1 los bits que están a 1 en \$a o \$b.

a ^ b → **O Exclusivo**: Se ponen a 1 los bits que están a 1 en \$a o \$b, pero no en ambos.

~ a → **No**: Se invierten los bits (se cambian 1 por 0 y viceversa.)

a << b → **Desplazamiento a Izquierda**: Desplaza \$b posiciones a la izquierda todos los bits de \$a.

a >> b → **Desplazamiento a Derecha**: Desplaza \$b posiciones a la derecha todos los bits de \$a.

Precedencia de operadores

Js

()	++ --	!	* / %
Parentesis	Incremento / decremento	Negación (lógico)	Producto / división
+ - .	<< >>	< <= > >= <>	== != === !==
Suma Resta /Concatenación	Despalzamiento de bits	Comparación	Comparación
&	^		&&
Y (Bits)	Bits	O (Bits)	Y (lógicos)
	? :	= += -= *= /= .= %= &= != ^= <<= >>=	
O (lógicos)	Ternario	Asignación	

Ordenados linealmente

Ejercicio (3d)

Operadores: A partir de las variables var1 a var6, creadas por el usuario y con los siguientes valores: true, false, "holá", "adios", 5, 2;

1. Determinar cual de los dos elementos de texto es mayor
2. Utilizando exclusivamente los dos valores booleanos, determinar los operadores necesarios para obtener un resultado true y otro resultado false
3. Determinar el resultado de las cinco operaciones matemáticas realizadas con los dos elementos numéricos

Objetivo: Continuar familiarizándonos con el uso de los operadores relacionales, lógicos y aritméticos.

Alternativa, en vez de las seis variables, podría utilizarse un array con los seis valores

[Desarrollos\JavaScript\JS05b_Operadores.html](#)



Control de flujo

jueves, 18 de mayo de 2017 11:45

- Control de flujo
 - i. Condiciones
 - ii. Iteraciones

Fundamentos. Control de flujo

OPCIONES

```
IF <condición>
  ELSE IF <condición>
  ....
  ELSE.
ENDIF.
```

ITERACIONES

```
DO
  .... EXIT
ENDDO
```

```
DO n TIMES
ENDDO
```

```
CASE <variable>
  WHEN <valor 1>
  ....
  WHEN <valor n>
  WHEN OTHERS
ENDCASE
```

Condicionadas

```
WHILE <condición>
ENDWHILE.
```

Control de flujo. Opciones (1)

Js

```
if(<condición>
    {...}
  else if(<condición>
    {...}
  else
    {...}
```



```
if(edad < 12) {
  alert("Todavía eres muy pequeño");
}
else if(edad < 19) {
  alert("Eres un adolescente");
}
else if(edad < 35) {
  alert("Aun sigues siendo joven");
}
else {
  alert("Piensa en cuidarte un poco más");
}
```

Control de flujo. Opciones (2)

Js

```
switch(<variable>)
{
    case <valor 1>:
        ....
        break
    case <valor n>:
        ....
        break;
default:
    ....
    break;
}
```

Para realizar comprobaciones múltiples y tomar decisiones complejas, si estas dependen siempre de la misma variable, el repetido uso de if es redundante y puede sustituirse mediante la estructura de control switch

- La variable siempre la misma se indica al principio
- Cada case corresponde a un valor posible de dicha variable
- Si el case no incluye un break continuaran evaluándose los siguientes valores
- La inclusión de default es opcional
- La evaluación sigue el orden en que se escribe el código, por lo que dicho orden es determinante

Ejemplo de switch

En el ejercicio realizado anteriormente para distinguir **Pares o Nones**, podemos sustituir el segundo if por una estructura switch, más eficaz.

```
function muestraDatos(codigo) {  
    var cTexto;  
    switch (codigo) {  
        case 0:  
            cTexto = TEXT1 + PAR;  
            break;  
        case 1:  
            cTexto = TEXT1 + IMPAR;  
            break;  
        case -1:  
            cTexto = TEXT2;  
            break;  
    }  
    document.write(cTexto);  
}  
// Fin de la función muestraDatos  
  
muestraDatos(parImpar(prompt(Q1))  
)  
  
function parImpar(numero) {  
    if (isNaN(parseInt(numero))) {  
        return -1; // No es un número  
    } else {  
        return (numero % 2);  
        // 1 si Numero impar & 0 Numero par  
    }  
} // Fin de la función parImpar  
  
function main() {  
    const Q1 = "Introduce un número entero",  
          TEXT1 = "El número introducido es ",  
          TEXT2 = "Eso no es un número",  
          PAR = "<b>par</b>",  
          IMPAR = "<b>impar</b>";  
}
```

Control de flujo. Opciones (3)

Js

? → operador ternario condicional

<condición> ? <expresión> : <expresión>



Devuelve expr2 si expr1 y expr3 si !expr1

```
edad < 12 ? "Todavía eres muy pequeño" : "Ya no  
eres tan pequeño, piensa un poco";
```

Es una forma muy compacta de construir un *if / else*
gracias al uso del **operador ternario condicional**.

Ejercicio (5)

Pares o Nones: Recoger un número, mediante document.write(prompt('Escribe un número')) o como un campo input, e indicar si se trata de un valor par o impar.

desarrollos\javascript\JS06_Pares_Nones.html

Objetivo: Utilizar bucles de control condicionales (**if**).
Anticipar el concepto de control de errores (e.g. utilizando la función isNaN)

Control de flujo. Iteraciones Incondicionales

Js

```
for(inicialización; condición; actualización) {  
    ...  
}  
    for(i=1; i <=n; ++i)  
    {  
        ....  
    }
```



```
var mensaje = "Hola, estoy dentro de un bucle";  
  
for(var i = 0; i < 5; i++) {  
    alert(mensaje);  
}  
  
var dias = ["Lunes", "Martes", "Miércoles", "Jueves",  
"Viernes", "Sábado", "Domingo"];  
  
for(var i=0; i<7; i++) {  
    alert(dias[i]);  
}
```

Ejercicio (8)

Factoriales: Utilizando la estructura for, crear un script que calcule el factorial de un número entero, que introduciremos mediante `document.write(prompt('Escribe un número'))` o como un campo input.

Objetivo: Utilizar bucles de control mediante iteraciones incondicionales (for).

desarrollos\javascript\JS07b_Factorial_1.html

NOTA: El factorial de un número entero n es una operación matemática que consiste en multiplicar todos los factores entre n y 1, es decir $n * (n-1) * (n-2) * \dots * 1$. Así, el factorial de 5 (escrito como 5!) es igual a:
 $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Factoriales como ejemplo de patrones

Los **factoriales** pueden calcularse mediante **recursividad**: una forma de crear una iteración de forma explícita mediante una función que **se llama a si misma**.

El factorial de un número entero n es un ejemplo perfecto de recursividad porque su propia definición matemática puede replantearse en esos términos, dado que $n! = n * (n-1)!$

Así, el factorial de 5 puede descomponerse como

$$5! = 5 \times 4! = 5 \times 4 \times 3! = 5 \times 4 \times 3 \times 2! = 5 \times 4 \times 3 \times 2 \times 1! = 120$$

Los **factoriales** pueden calcularse empleando un patrón de *callback*: la función que muestra los datos recibe como parámetro la función *callback* con la que va a realizar los cálculos

Sentencias *break* y *continue*

Js

permiten manipular el comportamiento normal de los bucles *for*

break permite terminar de forma abrupta un bucle

continue permite saltarse una repetición del bucle, pasando a la siguiente.

```
var cadena = "En un lugar de la Mancha...";  
var letras = cadena.split("");  
var resultado = "";  
  
for(i in letras) {  
    if(letras[i] == 'a') {break; }  
    else {resultado += letras[i]; }  
}  
alert(resultado); // muestra "En un lug"
```

usando *continue*, el resultado sería
"En un lugr de l
Mnch..."

Control de flujo.

Iteraciones Condicionadas

Js

```
while (<condición>)
{ .....
}
do {
.....
} while (<condición>)
```

bucles que se ejecutan ninguna o más veces, dependiendo de la condición indicada.

bucles que se ejecutan una o más veces, ya que lo hacen al menos la primera vez.

```
var resultado = 0;
var numero = 100;
var i = 0;
while(i <= numero) {
    resultado += i;
    i++;
}
alert(resultado);
```

```
var resultado = 1;
var numero = 5;
do {
    resultado *= numero;
    numero--;
} while(numero > 0);

alert(resultado);
```

Ejercicio (6)

Media aritmética (y otros estadísticos).

Realizamos una función que admita n parámetros y calcule su media aritmética

Nota: la media aritmética es la suma de los datos dividido por el número de valores

Objetivo: Familiarizarnos con el uso de **arguments** para determinar el número de parámetros.

Ejercicio (7)

Los números primos del 1 al 100. Presentar sucesivamente los números primos entre 1 y 100. Un numero primo es cualquier número mayor que 1 que no tiene divisores exactos a excepción de 1 y el mismo. Dicho en forma algorítmica, un número es primo si dividido por todos los que son menores que el y mayores que uno, no resulta ninguna operación que de como módulo (resto) 0.

desarrollos\JavaScript\JS07_Primos_1.html

Objetivo: Utilizar bucles de control condicionales (**if**), incluyéndolos dentro de un bucle **for**.

Ejercicio (9)

Serie de Fibonacci: Utilizando la estructura while o do while, crear un script que, después de solicitar su límite superior, calcule y muestre por pantalla la serie de Fibonacci.

desarrollos\javascript\JS08_Fibonacci_1.html

Ampliación:

- Utilizamos un patrón **callback** para pasar la función que realiza el cálculo
- en lugar de `prompt('....')`, introduciremos el límite de la serie con un **campo input**, y pasaremos el script a una función.

Objetivo: Utilizar bucles de control mediante iteraciones incondicionales (while / do while).

NOTA: la serie de Fibonacci comienza por 0 y 1 y continua obteniendo cada término como la suma de los 2 anteriores

Excepciones.

Js

Las excepciones son los errores ocasionados durante la ejecución de un programa

- En JS es frecuente que los errores permitan que el programa continúe, con resultados más o menos imprevisibles
- En cualquier caso, el sistema reacciona ejecutando un fragmento de código que muestra un mensaje de error o de aviso por la **consola**

```
<script>
    var x = noExiste();
</script>
```

Intentamos ejecutar una función inexistente



Gestión de excepciones.

Js

Es la creación de un sistema propio de la aplicación que incorpora una determinada reacción ante determinadas excepciones.

Se basa en los bloques de código creados con **try, catch, throw**

La instrucción **try** delimita un bloque de código y captura cualquier excepción que se produzca, almacenándola en memoria para pasársela a la cláusula **catch**.

La cláusula **catch** define un parámetro de tipo objeto error que recogerá los datos de la excepción.

```
try {  
    var x = noExiste();  
}  
catch(err) {  
    document.write("Error: " + err + ".");  
}
```

Excepciones provocadas.

Js

La instrucción *throw* permite desencadenar (*raise*) una excepción en respuesta a cualquier circunstancia elegida por el desarrollador.

throw va seguido de una expresión que puede ser un literal con el mensaje del error o un objeto literal con el nombre del error y el correspondiente mensaje.

- si se encuentra dentro de un bloque *try*, pasa el control a la cláusula *catch*, incluyendo el objeto error. La cláusula será la que actuó en función del error, en su caso abandonando la función.
- si *throw* no se encuentra dentro de un bloque *try*, se abandona la función y el control pasa a la cláusula *catch* del bloque *try* de la función que inicialmente hizo la llamada.

Excepciones provocadas. Ejemplo

```
function myFunction()
{
    var y = document.getElementById("mess");
    y.innerHTML = "";
    try {
        var x = document.getElementById("demo").value;
        if( x == "" ) throw "empty";
        if( isNaN(x) ) throw "not a number";
        if( x > 10 ) throw "too high";
        if( x < 5 ) throw "too low";
    }
    catch(err) {
        y.innerHTML = "Error: " + err + ".";
        return;
    }
}
```

Ejercicio (10)

Excepcion: Replantear el ejercicio del cálculo del factorial, en su versión que separa calculo y presentación, utilizando control de excepciones para las circunstancias en las que no es posible calcular el factorial.

[desarrollos\javascript\JS10_Factorial_Expcion.html](#)

Objetivo: Utilizar excepciones como mecanismo de validación y respuesta a los errores provocados por el usuario

Instrucciones: Recapitulación.

JS

Instrucciones de asignación

- var

Excepciones

- try
- catch
- throw

Block: no crea un ámbito diferente

Instrucciones disruptivas

- break
- return
- throw

Prefijo opcional label: para los bloques de instrucciones y su interacción con break

Instrucciones complejas

- block
- if
- switch
- for
- while
- do

Modificación del curso de ejecución por instrucciones disruptivas

Datos.

Tipos de datos

- Number
- String
- Boolean
- Undefined
- Objet; Function

Formas de almacenamiento

- Primitivos o elementales
- Referencias

Funciones

Valores

Variables y constantes

- var
- let y const (E6)
- Constantes predefinidas

Operadores y expresiones

- Casting automático
- Operadores

Control de flujo

- Condiciones
- Iteraciones

Funciones

jueves, 18 de mayo de 2017 11:44

Funciones. Declaración

son 100% objetos, creados por el constructor Function

```
var suma = new Function("a", "b", "return a+b")
```

Argumentos o
parámetros formales

Valor de retorno
(Opcional; en su ausencia se
devuelve el valor undefined)

Sin embargo, las funciones se crean normalmente
declarándolas mediante la palabra reservada function

```
function suma(a,b) { return a+b }
```

Argumentos o
parámetros
formales

Valor de retorno

Como veremos, Las funciones (por ser objetos)
también pueden ser declaradas como variables

Llamada a Funciones

La correspondiente llamada (o invocación) a la función sería

```
var resultado = suma(2, 3);
```

Parámetros reales

```
var resultado = suma(2*6, 3-4*3+4);
```

en el segundo caso, definido mediante operaciones
que se evalúan antes de pasarlos a la función

Técnicamente, la llamada a la función se
debe al operador de invocación ()

Funciones: declaración y llamada

Js

La **llamada** (o invocación) a la función puede ser anterior a su **declaración**. Esto se debe al proceso conocido como alzamiento: las declaraciones de variables y funciones son **alzadas** al principio por el interprete de JS (También denominado visibilidad sintáctica)

```
foo(); ←  
function foo(){  
    console.log('Se ejecutó la función!');  
}
```

La función se ejecuta sin problema, aunque está declarada posteriormente

Cuando una función se crea mediante asignación a una variable, el proceso de alzamiento no se produce

Funciones: ejemplo

Js

Definición de la función: en un fichero externo, en la cabecera, antes de invocarla o despues de invocarla

```
function calculaPrecioTotal (precio) {  
    var impuestos = 1.16;  
    var gastosEnvio = 10;  
    var precioTotal = ( precio * impuestos ) + gastosEnvio;  
    return precioTotal;  
}
```

Llamada a la función: en el punto donde queremos que se ejecute su código. Siempre con (), incluso si no hay argumentos.

```
var precioTotal = calculaPrecioTotal (23.34);
```

Ejercicio (4a)

Funciones. Realizamos 4 funciones que respectivamente:

- Escribe en pantalla
- Realiza las 4 operaciones (S,R,P,D)
- Convierte Km a Millas
- Convierte °Farenheit en °Celsius

El resultado será el que se muestra

El valor de n1 es: 60.5

El valor de n2 es: 20

El valor de varSuma es: 80.5

El valor de varResta es: 40.5

El valor de varProducto es: 1210

El valor de varDivision es: 3.025

El valor de n1 en millas aprox. es: 37

El valor de n1 °Farenheit en °Celsius es: 15

desarrollos\javascript\JS09_Funciones.html

Objetivo: Familiarizarnos con el uso de funciones, el paso de parámetros y la devolución de valores..

Funciones: tipo y número de parámetros

Js

Ante la ausencia de tipos claramente definidos

No es posible asegurar que los tipos de los parámetros sean adecuados para las operaciones que realiza la función .

Si hay más parámetros reales que los definidos en la función, se ignoran los que sobran

```
function suma(a, b) {  
    return a+b;  
}
```

```
var resultado = suma(2, 3, 4);
```

Si hay menos parámetros reales que los definidos en la función, los parámetros formales correspondientes a los que faltan se inicializan como *undefined*.

```
var resultado = suma(2);
```

Se ignora el 4

el parametro b = *undefined*

El objeto arguments en las funciones

Js

Dentro de cualquier función se genera automáticamente el array **arguments**

→ contiene todos los parámetros con los que se ha invocado a la función

la propiedad `arguments.length` devuelve el número de parámetros con los que se ha llamado a la función.

se puede acceder directamente a cualquier parámetro mediante la notación tradicional de los *arrays*

la propiedad `callee` hace referencia a la función que se está ejecutando, permitiendo, por ejemplo, mostrar su código fuente, o el número de parámetros que espera la función (`arguments.callee.length`)

Ejemplo: arguments

```
function mayor() {  
    var elMayor = arguments[0];  
    for(var i=1; i<arguments.length; i++) {  
        if(arguments[i] > elMayor) {  
            elMayor = arguments[i];  
        }  
    }  
    return elMayor;  
}
```

```
var variable1 = mayor(1, 3, 5, 8);  
var variable2 = mayor(4, 6, 8, 1, 2, 3, 4, 5);
```

La función no define ningún argumento a priori

La referencia a los argumentos siempre se hace mediante el objeto arguments[]

Es posible invocar la función con cualquier número de argumentos

Ámbito de las variables

Js

Hasta ES 6

Las variables poseen un alcance o ámbito (**scope**) al momento de su declaración indicando donde la puedo utilizar.

Las variables definidas fuera de cualquier función siempre se transforman automáticamente en **variables globales** y están disponibles en cualquier punto del programa

No existe, como en otros lenguajes, un alcance de variable por bloque (*block-scope*), como podría ser un if, for, while, etc .
Por tanto , en JavaScript el alcance es por función.

```
var cTexto;  
for(var i=0; i<100; i++){  
    cTexto = 'algun valor';  
}
```

```
for(var i=0; i<100; i++){  
    var cTexto;  
    cTexto = 'algun valor';  
}
```

Variables locales

Js

Las variables definidas dentro de una función pueden ser de dos tipos:

- las variables que se declaran mediante **var** son **variables locales**, que solo existen dentro de dicha función

Si una función define una variable local con el mismo nombre que una variable global, las variables locales prevalecen

La recomendación general es definir como variables locales todas las variables que sean de uso exclusivo para realizar las tareas encargadas a cada función.

- Los variable que **NO** son declaradas mediante **var** , se transforman automáticamente en **variables globales**. o son referencia a variables globales que ya existían.

Variables globales y locales. Ejemplo

Js

```
var cTexto1 = "valor inicial";
var cTexto2 = "valor inicial";
```

Variable local

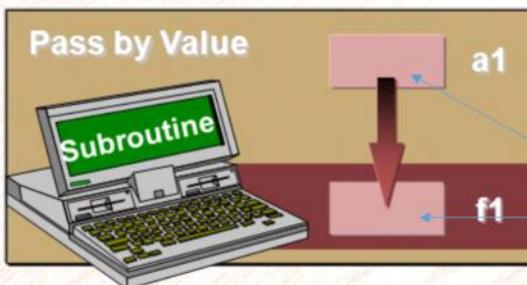
```
function unaFuncion () {
    var cTexto1
    cTexto1 = "Cambio desde dentro";
    cTexto2 = "Cambio desde dentro";
}
```

Variable global,
accedida desde la función

```
unaFuncion();
console.log(cTexto1) // devuelve "valor inicial"
console.log(cTexto2) // devuelve "Cambio desde dentro"
```

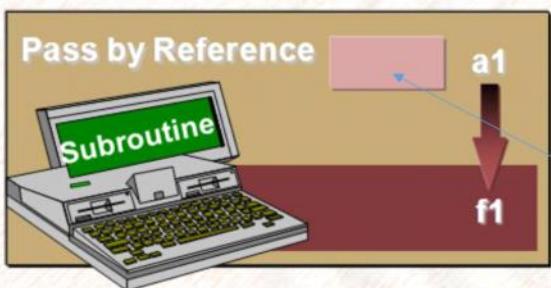
Fundamentos.

Paso de parámetros



Paso por valor

El parámetro formal hace referencia a un área de memoria independiente



Paso por referencia

Los parámetros real y formal hacen referencia a la misma área de memoria

Parámetros en JavaScript

Js

Sabemos que el acceso a las variables puede ser por valor (tipos primitivos) o por referencia (tipos referenciados u objetos)

[Repetir concepto](#)

Sin embargo, el paso de parámetros es SIEMPRE por valor

- para los **tipos primitivos** esto supone que no existe paso por referencia
- en los **tipos referenciados (objetos)**, el parámetro formal crea una nueva referencia al objeto original, lo que implícitamente supone siempre un paso por referencia.

Parámetros y tipos primitivos

JS

El acceso a las variables es por valor

El paso de parámetros es por valor

```
function addTen(num)
{
    num += 10;
    return num;
}
```

```
var count = 20;
var result = addTen(count);
```

```
alert(count); //20 - no
change
alert(result); //30
```

3

num es un parámetro local
que recibe el **valor** de count

1

count es una variable global
a la que se le asigna un **valor**

2

se pasa a la función
el **valor** de count

4

result almacena el **valor**
devuelto por la función

Parámetros y tipos referenciados

JS

El acceso a las variables es por referencia

El paso de parámetros es por valor

```
function setName(obj)
{
    obj.name = "Nicholas";
}

var person = new Object();
setName(person);

alert(person.name);
//Nicholas
```

3 obj es un parámetro local que recibe el valor de la referencia person, con lo que se crea una 2^a referencia a un único objeto

1 person es una variable global que recoge una referencia

2 se pasa a la función el valor de la referencia

4 el valor del objeto a cambiado; la función ha manipulado el objeto, u no es necesario que devuelva ningún valor

Ejercicio (4b)

Funciones, ámbitos y parámetros.

1. Creamos una función sin parámetros que sume dos números (accediendo a las variables globales) y almacene el resultado en una variable global (sin return)
2. Creamos una función con dos parámetros, que sume dos números y almacene el resultado en una variable global (sin return)
3. Creamos la función estándar, que suma dos parámetros y devuelve el resultado en un return.
4. Creamos una función que reciba como parámetro un objeto {var_1: 12, var_2: 5, resultado: 0}, sume los dos primeros atributos y almacene el resultado en el tercero.

Objetivo: Comprobar el ámbito de las variables y el paso de los parámetros.

Funciones: declaración mediante asignación

Js

Las funciones (como objetos que son) también pueden ser declaradas como variables

```
var suma = function (a, b) {  
    return a+b;  
}
```

Una forma más "estilo objetos", al parecerse a la forma de declarar los métodos

En realidad se le está **asignando** a la variable una **función anónima**

Igualmente podría hacerse la declaración asignando a la variable una función con nombre

```
var suma = function op (a, b) {  
    return a+b;  
}
```

El **nombre** op () es interno a la función suma, por lo que **no es utilizable**

Funciones anónimas

Como acabamos de ver, se puede definir una función con una expresión que **no incluye el nombre de la función**, por lo que se conocen como funciones anónimas o literales

```
function(a, b) { return a+b; }
```

Se declara una función
suma, sin definirla con un
nombre específico

Cuando la asignamos a una variables, el nombre de esta pasa a ser la referencia a la función

```
var resultado = function(a, b) { return a+b; }
```

Se utilizan mucho como parte de los objetos, al crear **métodos**,
y en frameworks como **JQuery**

Funciones y asignación

Js

Las funciones son **objetos** de pleno derecho:

- pueden asignarse a variables, propiedades, parámetros, en cualquier momento, después de haber sido declaradas (de cualquier forma)
- hay que distinguir la asignación de una función frente a la asignación de su resultado

```
var conducir = function(persona, coche) {  
    return (persona + " conduce un " + coche);  
};
```

```
var x = conducir; // asigna a x el código de la función  
x('José', 'Ferrari'); => 'José conduce un Ferrari'  
x() => 'undefined conduce un undefined'
```

```
var y = conducir(); // asigna a y el resultado de invocar  
la función  
y => 'undefined conduce un undefined'
```

Funciones y patrón "Callback"



Callback
Functions

una forma de usar las
funciones, pasándolas como
argumento a otra función

Pueden ser tanto funciones con nombre como anónimas o
métodos (veremos que con algunas particularidades)

Usos

- Evitar que 2 funciones sucesivas realicen el mismo bucle e.g. procesando un array
- Permitir asignar funciones como manejadores de eventos asíncronos
- Asignar funciones a *timeouts*, como otra forma de asíncronía

Funciones "Callback". Ejemplos

```
function one() { return 1;}  
function two() { return 2;}
```

```
function invoke_and_add(a, b){  
    return a() + b();  
}
```

```
invoke_and_add(one, two);
```

```
invoke_and_add(function(){return 1;},  
function(){return 2;})
```

Funciones anidadas

Js

Una función puede contener en su interior otras funciones anidadas:

```
function sumaCuadrados(a, b) {  
    function cuadrado(x) { return x*x; }  
    return cuadrado(a) + cuadrado(b);  
}  
  
sumaCuadrados(4,5) // 41
```

Al anidar funciones, las funciones internas (*inner*) no son accesibles desde fuera de la función que las contiene, por lo que se pueden denominar funciones privadas (*private*)

Si el retorno es la función sin ejecutar, la invocación de la función exterior sería nombreFuncion()()

Funciones auto-invocadas

JS

Se puede auto invocar una función (generalmente anónima) como forma de crear un código que se ejecuta directamente al tiempo que define su propio espacio de memoria

```
( function(p1, p2){  
    //código de tu función  
} (var1, var2) );
```

Parámetros formales;

Parámetros reales;

```
(function(a, b) { return a+b; } (5,6))
```

Se ejecuta directamente la función suma, de ejemplos anteriores con los parámetros reales suministrados

Cierres o closures

Js

Una función que encapsula un conjunto de definiciones locales : variables, otras funciones (por extensión objetos). Se basa por tanto en el **anidamiento de funciones**.

permite que la función interna
"escape" de la función padre → se hace accesible desde fuera

la función interna sigue
teniendo acceso al ámbito de
la función padre → sigue ligada al ámbito (*scope*)
de la función padre después
de que esta haya retornado

El ámbito de visibilidad de la función externa se utiliza para crear
el equivalente a un **módulo**.

Las funciones internas pueden ser *setters* y *getters* de variables
locales, no accesibles directamente desde fuera

Cierres o closures: patron 1



Una función incluye una función interna y la devuelve

```
var accesoClosure = function() {  
    var ... // variables locales  
    function funcionLocal () {...}  
    return funcionLocal  
}();
```

- solo son accesibles a través de una interfaz (función u objeto) que corresponde al parámetro de retorno de la función
- el resultado de su ejecución se asocia a una variable, cuyo nombre pasa a ser equivalente al interfaz: accesoClosure equivale a funcionLocal, pero desde el ámbito de visibilidad exterior: accesoClosure() ejecuta la función local

Cierres o closures: patron 2



Una función incluye una función interna y se la asigna a una variable global

```
var accesoClosure
function() {
    var ... // variables locales
    function funcionLocal () {...}
    accesoClosure = funcionLocal
} ();
```

- solo son accesibles a través de una interfaz (función u objeto) que corresponde al parámetro de retorno de la función
- la función interna se asocia a una variable global, cuyo nombre pasa a ser equivalente al interfaz: accesoClosure() equivale a funcionLocal(), pero desde el ámbito de visibilidad exterior

Cierres o closures. Ejemplo

Js

```
var enteroUnico = function () {  
    var contador = 0;  
    function contar () { return  
        contador++; };  
    return contar;  
} ();  
  
enteroUnico() => 0  
enteroUnico() => 1
```

El mismo código, escrito más compacto

```
var enteroUnico = function () {  
    var contador = 0;  
    return function () { return contador++; };  
} ();
```

closure y autoejecución

Js

La creación de su propio espacio de memoria en las funciones hace que se puedan utilizar para aislar un código fuente y evitar colisiones de variables o espacios de nombres en general.

Este efecto de los cierres (clorure) y junto con las funciones anónimas auto-invocadas es la base del patrón denominado IIFE (*Immediately-invoked function expression*)

```
(function(){
    //código de tu función
})();
```

El código se ejecuta directamente, como si no hubiera función, pero en su propio espacio de memoria

Los cierres son también la base de patrones de diseño que generan un ámbito privado en los objetos definidos literalmente



Una promesa representa el resultado eventual de una operación.
Se utiliza para especificar que se hará cuando esa eventual operación de un resultado de éxito o fracaso.

Promesas

JS

Un objeto promesa representa un valor que todavía no está disponible pero que lo estará en algún momento en el futuro

Permiten escribir código asíncrono de forma más similar a como se escribe el código síncrono:

- La función asíncrona retorna inmediatamente y
- ese retorno se trata como un proxy cuyo valor se obtendrá en el futuro

El API de las promesas en Angular corresponde al servicio **\$q**

la biblioteca Q desarrollada por **Kris Kowal**

<https://github.com/kriskowal/q>



Promesas: \$q

JS

```
function getPromise()
```

```
    var deferred=$q.defer();
```

crea una promesa

```
    deferred.resolve()  
    deferred.reject()
```

resuelve la promesa en un sentido
u otro al cabo del tiempo

```
    return deferred.promise
```

devuelve la promesa

```
var promise = getPromise();
```

```
promise.then(successCallback,failureCallback,notifyCallback);  
promise.catch(errorCallback)  
promise.finally(callback)
```

promise.catch(errorCallback)

promise.finally(callback)



Callbacks anidados

sábado, 14 de octubre de 2017 13:52

```
function msgAfterTimeout (msg, nombre, tiempo, cb) {
    setTimeout(function () {
        cb(msg, nombre);
    }, tiempo);
};
```

Se declara una función asincrónica

En ella se ejecuta la función recibida como callback dentro del setTimeout. Para poder pasárle parámetros hay que incluirla en una función anónima

Se invoca la función asincrónica

```
msgAfterTimeout("", "Pepe", 100,
    function (msg, nombre) {
        let saludo = (`${msg} Hola ${nombre}!`);

        msgAfterTimeout(saludo, "Juan", 200,
            function (msg, nombre) {
                let saludo = (`${msg} Hola ${nombre}!`)

                console.log(`Saludo después de 0,3 seg: ${saludo}`);
            } // Fin de la función callback
        ); // Fin de la llamada a msgAfterTimeout
    } // Fin de la función callback
); // Fin de la llamada a msgAfterTimeout
```

función enviada como callback

función enviada como callback

operación con el resultado acumulado de los dos callbacks

Se invoca nuevamente la función asincrónica

Promesas en ES6

jueves, 10 de agosto de 2017 20:35

Implementación new Promise

el objeto promesa recibe como parámetros dos funciones:

- La función "resolve": se ejecutará cuando queramos finalizar la promesa con éxito.
- La función "reject": se ejecutará cuando queramos finalizar una promesa informando de un caso de fracaso.

```
function hacerAlgoPromesa (){  
    return new Promise (function (resolve, reject) {  
        console.log ('hacer algo que ocupa un tiempo...');  
        setTimeout (resolve(), 1000);  
    })  
}
```

En este caso la promesa siempre se resuelve correctamente; la función admite como parámetro los datos que la promesa deba retornar

Utilización

a la función que retorna el objeto promesa se le encadenan el método then con dos funciones como parámetros,

- la función que se ejecutará cuando la promesa haya finalizado con éxito.
- la función que se ejecutara cuando la promesa haya finalizado informando de un caso de fracaso.

```
hacerAlgoPromesa()  
.then (  
    function (){ console.log (' la promesa terminó.'),  
    function (){ console.log (' la promesa fracasó.')}  
)
```

Alternativamente puede utilizarse el método catch para declarar la función que se ejecutara cuando la promesa haya finalizado informando de un caso de fracaso.

```
hacerAlgoPromesa()  
.then (function (){ console.log (' la promesa terminó.'))  
.catch(function (){ console.log (' la promesa fracasó.'))
```

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

Ejemplo de promesas en ES6

sábado, 14 de octubre de 2017 14:43

```
function msgAfterTimeout (msg, nombre, tiempo) {  
    return new Promise((resolve, reject) => {  
        setTimeout(  
            () => resolve(`${msg} Hola ${nombre}!`),  
            tiempo  
        )  
    })  
}
```

Función que crea y devuelve un **objeto promesa**

En este caso, la promesa siempre se resuelve correctamente, creando un mensaje de saludo a un usuario

Utilización de las promesa, encadenando las llamadas a ellas

```
msg almacena el resultado del primer proceso asincrónico  
  
msg almacena los sucesivos resultados de los procesos asincrónico  
  
msgAfterTimeout("", "Pepe", 100)  
.then((msg) =>  
    msgAfterTimeout(msg, "Juan", 200))  
.then((msg) => {  
    console.log(`Saludo después de 0,3 seg: ${msg}`)  
})
```

operamos finalmente con *msg*

```
MENSALES  
Saludo después de 0,3 seg: Hola Pepe! Hola Juan!  
PS D:\Desarrollo\Front_End_alce65\Angular\angular_4_2017\02_tecnologias\ES6>
```

Asinc / Await

domingo, 9 de septiembre de 2018 12:05

Definido en ES2017 (ES8)

La declaración de función `async` define una función asíncrona, la cual devuelve un objeto `AsyncFunction`.

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  var result = await resolveAfter2Seconds();
  console.log(result); // expected output: 'resolved'
}

asyncCall();
```