

UT4. Estructuras definidas por el usuario en JavaScript

- 1) Estructuras de datos.
 - 1) Objeto Array.
 - 2) Arrays paralelos.
 - 3) Arrays multidimensionales.
- 2) Creación de objetos definidos por el usuario.
 - 1) clases
 - 2) objetos literales
 - 3) Ejemplo de objeto literal: JSON
- 3) Creación de funciones.
 - 1) Parámetros.
 - 2) Ámbito de las variables.
 - 3) Funciones anidadas.
 - 4) Funciones predefinidas del lenguaje.



1. Estructura de datos

Es una variable que te permite guardar **más de un valor**.

Los arrays son unidimensionales, pero si cada elemento del array contiene otro array tendremos un array bidimensional (matriz).

Se utilizan para guardar datos que van a ser accedidos **de forma aleatoria**. Si su acceso será secuencial es mejor utilizar listas.

Cada elemento es referenciado por la posición que ocupa (índice)

En JavaScript los arrays utilizan una **indexación base-cero(0)**. El primer elemento es el 0.

Existen arrays escalares (índice numérico) y asociativos (índice por claves).

1.1. Objeto array I

En JavaScript se utiliza mucho para guardar elementos de un documento HTML (por ejemplo los enlaces, imágenes, ...)

`document.links[0]` → **es el primer enlace del documento.**

Más ejemplos:

- Guardar coordenadas de una trayectoria.
- Guardar un listado de coches.

Para crear un array:

`var miArray = new Array ();` → con longitud 0 (`miArray.length`)

`var alumnos = new Array (30);` → con longitud 30 (`alumnos.length`)

Podemos crear un elemento más allá de la longitud, cambiando `length` a la nueva longitua

`alumnos[45]="Tomas";` → **`alumnos.length` ahora es 46.**

1.1. Objeto array II

Se puede crear un array así (se llama “array denso”):

```
sistemaSolar = new Array  
  
("Mercurio", "Venus", "Tierra", "Marte", "Jupiter", "Saturno", "U  
rano", "Neptuno");
```

También está permitido crear un array denso así:

```
sistemaSolar =  
["Mercurio", "Venus", "Tierra", "Marte", "Jupiter", "Saturno", "U  
rano", "Neptuno"];
```

Otra forma de crear un array (llamada objeto literal):

```
var datos = { "numero": 42, "mes": "Junio", "hola" :  
"mundo", 69 : "96" }; → es como un objeto.
```

Se recorre así: `datos["numero"]`

1.1. Recorrer un array I

Bucle for:


```
for (i=0;i<array.length;i++) {  
    sentencia con array[i];  
}
```

Bucle while:

```
var i=0;  
while (i < array.length) {  
    sentencia con array[i];  
    i++;  
}
```

1.1. Recorrer un array II

Método `forEach()`

```
var text="";  
  
function f1 (item,index){  
text+="El elemento nº "+index+" tiene valor "+item+"<br>";  
}  
  
sistemaSolar.forEach(f1); 
```

//text vale :

El elemento nº 0 vale Mercurio

El elemento nº 1 vale Venus

... .

1.1. Borrado de un elemento.

Con el operador delete:

```
elarray.length; // resultado: 8
```

```
delete elarray[5];
```

```
elarray.length; // resultado: 8
```



```
elarray[5]; // el tipo de datos es undefined
```

También se puede hacer algo equivalente así
(aunque el tipo de datos será otro)

```
elarray[5]=""; //el tipo de datos será string
```

```
elarray[5]=null; // el tipo de datos será object
```



1.1. Métodos de objeto array

concat() Une dos o más arrays, y devuelve una copia de los arrays unidos.

join() Une todos los elementos de un array en una cadena de texto separados por coma


pop() Elimina el último elemento de un array y devuelve ese elemento.

shift() Elimina el primer elemento de un array, y devuelve ese elemento.

push() Añade nuevos elementos al final de un array, y devuelve la nueva longitud.

unshift() Añade nuevos elementos al comienzo de un array, y devuelve la nueva longitud.

reverse() Invierte el orden de los elementos en un array.

 **slice(pos_ini[,pos_fin])**

sort([function]) Ordena los elementos del array alfabéticamente o aplicando una función. Modifica el array. Para ordenar números: `arr.sort(function(a, b){return a-b})`

 **splice()** Añade/elimina/sobreescribe elementos a un array. Modifica el array.

 **toString()** Convierte un array a una cadena y devuelve el resultado (como `join()`)

1.1. Métodos de objeto array – `slice()`

`slice(pos_ini[,pos_fin])` Selecciona una parte de un array y devuelve el nuevo array. Ambas posiciones pueden ser negativos para indicar desde el final, teniendo el último posición -1. El elemento que está en la posición `pos_fin` no se extrae. Devuelve el array extraído. El array original no se modifica.

Ejemplos:

```
var array=["a","b","c","d","e"]
```

```
array.slice(1) //dev. desde el elem 1 (el 1º es el 0) hasta el fin:["b","c","d","e"]
```

```
array.slice(1,2) //devuelve desde el elemento 1 hasta el elemento 2 (excluido): ["b"]
```

```
//devuelve desde el 4º por el final (el ultimo es el -1 no el 0), al  
elemento 2º por el final ["b","c"]
```

```
array.slice(-4,-2)
```

```
//devuelve desde el 3º empezando por el principio, al elemento 2º por el  
final ["c"]
```

```
array.slice(2,-2)
```



1.1. Métodos de objeto array – splice()



Sintaxis: `array.splice(index, howmany, item1,, itemX)`

Index: es la posición del array en la que se va a hacer la acción de añadir/borrar/sobreescribir.

howmany: es el número de elementos que van a ser eliminados. Si no se elimina ningún elemento valdrá 0. Si se van a sobreescribir elementos, se indica qué número de elementos van a ser sobreescritos.


item1, ..., itemX: son los elementos a añadir en el array, concretamente en la posición indicada en `index`.

Nota: realmente la acción de sobreescribir consiste en borrar elementos y añadir después.

1.1. Métodos de objeto array – splice()

Ejemplos:

//sustituye desde el elemento 1 en adelante 2 unidades, por "B" y "C" modifica a ["a","B","C","d","e"]

array.splice(1,2,"B","C") 

 //borra desde la posición 1 2 elementos, deja el array ["a","d","e"]

array.splice(1,2)

//añade "B" y "C" desde la posición 1, deja el array ["a","B","C","d","e"]

array.splice(1,0,"B","C")

//Borra desde la posición 1 2 elementos, y los sustituye por "X", deja el array ["a","X","d","e"]

array.splice(1,2,"X")

1.2. Arrays paralelos

Con dos o más arrays, que utilizan el mismo índice para referirse a términos homólogos.

Por ejemplo:

```
var profesores = ["Cristina", "Catalina", "Vieites", "Benjamin"];
```

```
var asignaturas=["Seguridad", "Bases de Datos", "Sistemas  
Informáticos", "Redes"];
```

```
var alumnos=[24,17,28,26];
```

Usando estos tres arrays de forma sincronizada, podemos saber que la profesora Cristina imparte Seguridad y tiene 24 alumnos.

1.3. Array multidimensionales



Si bien es cierto que en JavaScript los arrays son unidimensionales, podemos crear arrays que en sus posiciones contengan otros arrays u otros objetos. Podemos crear de esta forma arrays bidimensionales, tridimensionales, etc.

Ejemplo de array bidimensional:

```
var datosAlum = new Array();  
datosAlum[0] = ["Juan", "Perez", 22];  
datosAlum[1] = ["Luis", "Aragon", 20];  
datosAlum[2] = new Array("Ana", "Gomez", 19);  
datosAlum[3] = ["Antonio", "Martin", 21];
```

`console.table(datosAlum)`



`console.table()`

debugger eval code:1:9

(índice)	0	1	2
0	juan	perez	22
1	luis	aragon	20
2	Ana	Gomez	19
3	Antonio	Martin	21

`datos[3][2]` → será 21

`datos[1][1]` → será "Aragón"

1.3. Array anidados

Ejemplo de array anidados. Se trata de un array bidimensional, pero el elemento 3 de la 2ª dimensión (la cuarta columna) es a su vez otro array

```
var datosAlum = new Array();  
  
datosAlum[0] = ["Juan", "Perez", 22, ["DWECE", "DIW"]];  
  
datosAlum[1] = ["Luis", "Aragon", 20, ["DWES", "DAW"]];  
  
datosAlum[2] = ["Ana", "Gomez", 19, ["FOL", "ING"]];  
  
datosAlum[3] = ["Antonio", "Martin", 21, ["EMP", "BD"]];
```

```
console.table(datosAlum)
```

```
console.table()
```

debugger eval code:1:9

(índice)	0	1	2	3
0	juan	perez	22	► Array ["DWECE", "DIW"]
1	luis	aragon	20	► Array ["DWES", "DAW"]
2	Ana	Gomez	19	► Array ["FOL", "ing"]
3	Antonio	Martin	21	► Array ["Empr", "BD"]

datos[3][2] → será 21

//cuando accedemos a la cuarta columna como es un array a su vez, podemos indicar qué dato de ese array deseamos

datos[1][3][1] → será "DAW"

2. Creación de objetos definidos por el usuario.



Puedes crear tus propios objetos con propiedades y métodos.

2. Creación de objetos definidos por el usuario. Clases

Sintaxis: 

```
class ClassName {  
    constructor() { ... }    //método obligatorio  
    method_1() { ... }        //métodos opcionales  
    method_2() { ... }  
    method_3() { ... }  
}
```

/*IMP: El nombre de la clase comienza por mayúscula, para indicar que se debe instanciar con **new**

2. Creación de objetos definidos por el usuario. Clases

```
class Coche {  
  
    constructor(marca,combustible ) {    //método constructor es obligatorio  
  
        // Propiedades  
  
        // this será el objeto en el que se ha guardado la instancia creada de esta clase  
        // al ejecutar new.  
  
        this.marca =marca;  
        this.combustible = combustible;  
        this.cantidad = 0; //cantidad inicial de combustible  
    }  
  
    //Resto de método, que son opcionales  
    rellenarDeposito (litros) {  
        // Modificamos la propiedad cantidad de combustible  
        this.cantidad = litros;  
    }  
}
```

//Se utiliza así:

```
var auto=new Coche("Mercedes","diesel"); // Crear una instancia  
auto.marca // Para hacer referencia a la propiedad marca del objeto  
auto.rellenarDeposito(40); // Utilizar un método del objeto
```

2. Creación de objetos definidos por el usuario. Objetos literales.

Un literal es un valor fijo que se especifica en JavaScript. Un objeto literal será un conjunto, de cero o más parejas del tipo

nombre:valor o "nombre":valor

Ejemplo:

```
avion={  marca:"Boeing", modelo:"747", pasajeros:"450"  };
```

// también es válido poniendo el nombre de la propiedad entre "

```
avion={  "marca":"Boeing", "modelo":"747", "pasajeros":"450"  };
```

Es equivalente a:

```
var avion = new Object();
```

```
avion.marca = "Boeing";
```

```
avion.modelo = "747";
```

```
avion.pasajeros = "450";
```

2. Creación de objetos definidos por el usuario. Objetos literales.

Para acceder a una propiedad haremos:

Para referirnos desde JavaScript a una propiedad del objeto avión podríamos hacerlo con:

```
avion.modelo
```

```
avion["modelo"];
```

Para recorrer el objeto recordad que sería:

```
for (var prop in avion)
{
    document.getElementById("demo").innerHTML+=
        "La propiedad " + prop + " vale " + avion[prop];
}
```

2. Ejemplo objeto JSON (objeto literal)

<https://mdn.github.io/learning-area/javascript/ojs/json/superheroes.json>

Estos datos JSON, serán cargados en una variable, como por ejemplo:

```
var objJSON; // que contendrá { ..... } , el objeto JSON
```

Descargar y abrir con Bloc de notas para ver el objeto JSON

Observa que es un objeto literal.

Algunas propiedades del objeto literal como "members" son un array.

Los elementos del array "members" son objetos, a su vez.

Cada uno de estos objetos que forman el array "members" tienen una serie de propiedades.

Una de las propiedades de los objetos de "members" es "powers" que, a su vez, es otro array de string.

En definitiva, el objeto JSON, es un objeto, y una de sus propiedades es un array de objetos literales, a su vez, que tienen, a su vez, una propiedad que es otro array.

2. Ejercicio objeto JSON

1. Descarga el fichero JSON anterior en un fichero de texto.
2. Copia este objeto JSON en un fichero llamado ejJSON.js
3. Llama a ejJSON.js desde un documento HTML.
4. Muestra en un párrafo del documento HTML los siguientes datos:
 1. La fecha de creación del grupo de superhéroes.
 2. El número de miembros del grupo de superhéroes.
 3. La edad del superhéroe Eternal Flame (sin saber qué posición ocupa).
 4. El nombre del superhéroe que tiene el poder “Radiation blast”.
 5. El nombre del superhéroe con mayor número de poderes.

NOTA: para los apartados 3, 4 y 5 utiliza el método `forEach()`

2. Funciones I

Es un conjunto de acciones preprogramadas. Las funciones se llaman a través de eventos o bien mediante comandos desde nuestro script.

Permiten realizar tareas de una manera mucho **más organizada**, y además le permitirán reutilizar código en sus aplicaciones, y entre aplicaciones.



```
function nombreFunción ( [parámetro1]....[parámetroN] ) {  
    // Sentencias  
}
```

```
function nombreFunción ( [parámetro1]....[parámetroN] ) {  
    // Sentencias  
    return valor;  
}
```

// esta última función devuelve un valor con return que se recoge así:

```
var variable=funcion();
```

2. Funciones II

```
nombreFuncion( );
```

```
// Se ejecutaría las sentencias programadas dentro de la  
función sin devolver ningún valor.
```

```
variable=nombreFuncion( );
```

```
// la función ejecutaría las sentencias que  
contuviera y devolvería un valor que se asigna a  
la variable.
```

Asignar un nombre a una función que indique qué tipo de acción realiza. Suelen llevar un verbo (inicializar, calcular, borrar, ...)

Las funciones deben realizar **funciones muy específicas**. No deben realizar tareas adicionales a las inicialmente propuestas en esa función. Deben ser **lo más atómicas posible** para que se código sea lo más aprovechable posible.

2. Funciones III

Las funciones en JavaScript son objetos, y como tal tienen métodos y propiedades.



Un método, aplicable a cualquier función puede ser `toString()`, el cuál nos devolverá el código fuente de esa función.

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

`suma.toString()` → devuelve el código anterior.



`suma` → también devuelve el código anterior.

`suma.valueOf()` → también devuelve el código anterior

`suma(3,2)` → ejecuta el código que hay dentro de la función y devuelve un dato que es el resultado de la operación que realiza la función.

2.1. Parámetros



Son conocidos como **argumentos**. Permiten enviar datos entre **funciones**.

Para pasar parámetros a una función, tendremos que **escribir dichos parámetros entre paréntesis y separados por comas**.

Al definir una función que recibe parámetros, lo que haremos es, escribir los nombres de las variables que recibirán esos parámetros entre los paréntesis de la función.

```
function saludar(a,b) {  
  alert("Hola " + a + " y "+ b +".");  
}
```

a y b son los parámetros de la función.

2.1. Parámetros





Los parámetros de una función que sean de tipo Number, String o Boolean se pasan a la función por valor, no por referencia, o lo que es lo mismo son parámetros de entrada, no de entrada/salida, y por tanto, son tratados como variables locales a la función, por lo que aunque la función las modifique en su interior, al terminar la función las variables de los parámetros no van a variar.

Ejemplo: Ejercicio 4 de Ejercicios UT4.

[Ver imagen](#)

2.1. Parámetros:objetos

El contenido de la variable pasada como argumento a una función sí puede ser modificado por la función, en los los siguientes tipos de datos:

-  Un objeto definido por el usuario.
- Un objeto predefinido de Java Script alto nivel. 
- Un objeto predefinido de Java Script que no sea Number, Boolean o String.
- Un array.

Esto es así porque las propiedades de los objetos definidos por los usuarios, los elementos de un array, y las propiedades de una fecha son, en realidad, direcciones de memoria que apuntan a las zonas de memoria en las que está almacenadas las propiedades de estos objetos.

2.1. Parámetros. Ejemplos

```
<script>
var o1=new Number(3);
var d1=new Date();

function fNum(obj){
    obj=4;
    document.write("Dentro de funcion: ",obj,"<br>");
}

function fFecha(obj){
    obj.setDate(obj.getDate()-5);
    document.write("Dentro de funcion: ",obj.getDate(),"<br>");
}

document.write("<br>Funcionamiento de una funcion con parámetro un objeto Number<br>");
document.write("Fuera de funcion antes de llamarla: ",o1,"<br>");
fNum(o1);
document.write("Fuera de funcion despues de llamarla: ",o1,"<br>");

document.write("<br>Funcionamiento de una funcion con parámetro un objeto date<br>");
document.write("Fuera de funcion antes de llamarla: ",d1.getDate(),"<br>");
fFecha(d1);
document.write("Fuera de funcion despues de llamarla: ",d1.getDate(),"<br>");
</script>
```

2.1. Parámetros. Ejemplos

```
<script>
var a1=new Array("1","2","3");
var oDefUsu1={nombre:"Maite",apellido:"martinez"};

function fArray(obj){
    obj[2]="kkk";
    document.write("Dentro de funcion: ",obj.toString(),"<br>");
}

function fObjDefinidoUsu(obj){
    obj.nombre="otro";
    document.write("Dentro de funcion: ",obj.nombre,"<br>");
}

document.write("<br>Funcionamiento de una funcion con parámetro un objeto array<br>");
document.write("Fuera de funcion antes de llamarla: ",a1.toString(),"<br>");
fArray(a1);
document.write("Fuera de funcion despues de llamarla: ",a1.toString(),"<br>");

document.write("<br>Funcionamiento de una funcion con parámetro un objeto
predefinido<br>");
document.write("Fuera de funcion antes de llamarla: ",oDefUsu1.nombre,"<br>");
fObjDefinidoUsu(oDefUsu1);
document.write("Fuera de funcion despues de llamarla: ",oDefUsu1.nombre,"<br>");
</script>
```

2.3. Funciones anidadas I.

Podemos programar una función dentro de otra función.

```
function principalB() {  
    // Sentencias  
    function internaB1() {  
        // Sentencias  
    }  
    function internaB2() {  
        // Sentencias  
    }  
    // Sentencias  
}
```

2.3. Funciones anidadas II

Se aplican cuando:

- Tenemos una secuencia de instrucciones que **necesitan ser llamadas desde múltiples sitios dentro de una función.**
- Y esas instrucciones **sólo tienen significado dentro del contexto de esa función principal.**
Y por tanto, no tiene sentido definirla como una función global.

La función interna será privada o local a la función principal.

2.3. Funciones anidadas III

Ejemplo:

```
function hipotenusa(a, b) {  
    function cuadrado(x) {  
        return x*x;  
    }  
    return Math.sqrt(cuadrado(a) +  
cuadrado(b) );  
}
```


Función flecha

Existe una sintaxis más reducida para funciones. Se trata de la función flecha.

CUIDADO: El operador this no funciona igual dentro de una función flecha que en una función normal.

CUIDADO: No se puede utilizar como función constructora.

```
// Función tradicional
```

```
ejemplo=function (a) {
```

```
    return a + 100;
```

```
}
```

Función flecha

Ejemplos con funciones con varios argumentos y con una sola sentencia en el cuerpo de la función:

```
// Función tradicional  
ejemplo=function (a, b) {  
    return a + b + 100;  
}
```

```
// Función flecha  
ejemplo=(a, b) => a + b + 100;
```

Función flecha

Ejemplos con funciones sin argumentos y con una sola sentencia en el cuerpo de la función:

```
// Función tradicional (sin  
argumentos)
```

```
a = 4;
```

```
b = 2;
```

```
ejemplo=function () {  
    return a + b + 100;  
}
```

Función flecha

Del mismo modo, si el cuerpo tiene más de una sentencia introduce las llaves Más el "return":

```
// Función tradicional  
ejemplo=function (a, b) {  
    num = 42;  
    return a + b + num;  
}
```

```
// Función flecha. No se pueden
```

Función flecha

Y finalmente, en las funciones con nombre incluido en la definición:

```
// Función tradicional  
function sumar100 (a) {  
    return a + 100;  
}
```

```
// Función flecha  
sumar100 = a => a + 100;
```

2.4. Funciones globales (funciones de objeto window)

`decodeURI()` Decodifica los caracteres especiales de una URL excepto:

`decodeURIComponent()` Decodifica todos los caracteres especiales de una URL.

`encodeURI()` Codifica los caracteres especiales de una URL excepto: ,

`encodeURIComponent()` Codifica todos los caracteres especiales de una URL.

`escape()` Codifica caracteres especiales en una cadena, excepto: *

`unescape()` Decodifica caracteres especiales en una cadena, excepto: *

`eval()` Evalúa una cadena y la ejecuta si contiene código u operaciones.

`isFinite()` Determina si un valor es un número finito válido.

`isNaN()` Determina cuando un valor no es un número (mejor `Number.isNaN()`)

`Number()` Convierte el valor de un objeto a un número.

`String()` Convierte el valor de un objeto a un string

`parseFloat()` Convierte una cadena a un número real.

`parseInt()` Convierte una cadena a un entero.

2.2 Ámbito de las variables I

Las variables que se definen fuera de las funciones se llaman variables globales.

Las variables que se definen dentro de las funciones, con la palabra reservada `var` , se llaman variables locales.

El alcance de una variable global, se limita al documento actual que está cargado en la ventana del navegador o en un frame. Todas las instrucciones de tu script (incluidas las instrucciones que están dentro de las funciones), tendrán acceso directo al valor de esa variable.

2.2. Ámbito de las variables II

Una variable local será definida dentro de una función. En este caso, si que **se requiere el uso de la palabra reservada var** cuando definimos una variable local, ya que de otro modo, esta variable será reconocida como una variable global.

El alcance de una variable local está solamente dentro del ámbito de la función. Ninguna otra función o instrucciones fuera de la función podrán acceder al valor de esa variable.

2.2. **Ámbito de las variables**

- **Si se declara una variable dentro de una función con var, solo es accesible dentro de esa función.** El tiempo de vida de esta variable es desde que se declara hasta que la función acaba.
- **Si se declara una variable fuera de una función, es accesible también dentro de la función.** Por lo que pueden modificarla.
- **Si se crea una variable dentro de una función sin declararse (`x=3`) será considerada variable global por lo que también es accesible fuera de la función también.**
- **En HTML una variable global solo es global**

2.2. Variables de bloque

- **Se crean con let y se pueden usar dentro de un bloque de código.** Un bloque de código son las sentencias incluidas entre llaves o entre paréntesis (en un for).

- {

```
let x=2
```

```
}
```

```
// Aquí no se puede utilizar x
```

- {

```
var x=2
```

2.2. Variables de bloque

- ```
Var x=2;

{
 let x=5
 // Aquí x es igual a 5
}

// Aquí x es igual a 2
```

- **Si se declara una variable con let en el cuerpo principal (fuera de un bloque) su ámbito es el global.**

**Si se declara una variable local en una**

## 2.2. Variables de bloque

- **Con let no se puede redeclarar:**

- {

```
let x=2
```

```
let x=3
```

```
} //NO es válido
```

- **Se puede redeclarar con let una variable ya declarada pero en otro ámbito:**

```
let x=2;
```

## 2.2. Variables const

- **Para declarar variables cuyos valores no pueden cambiar se utiliza const.**

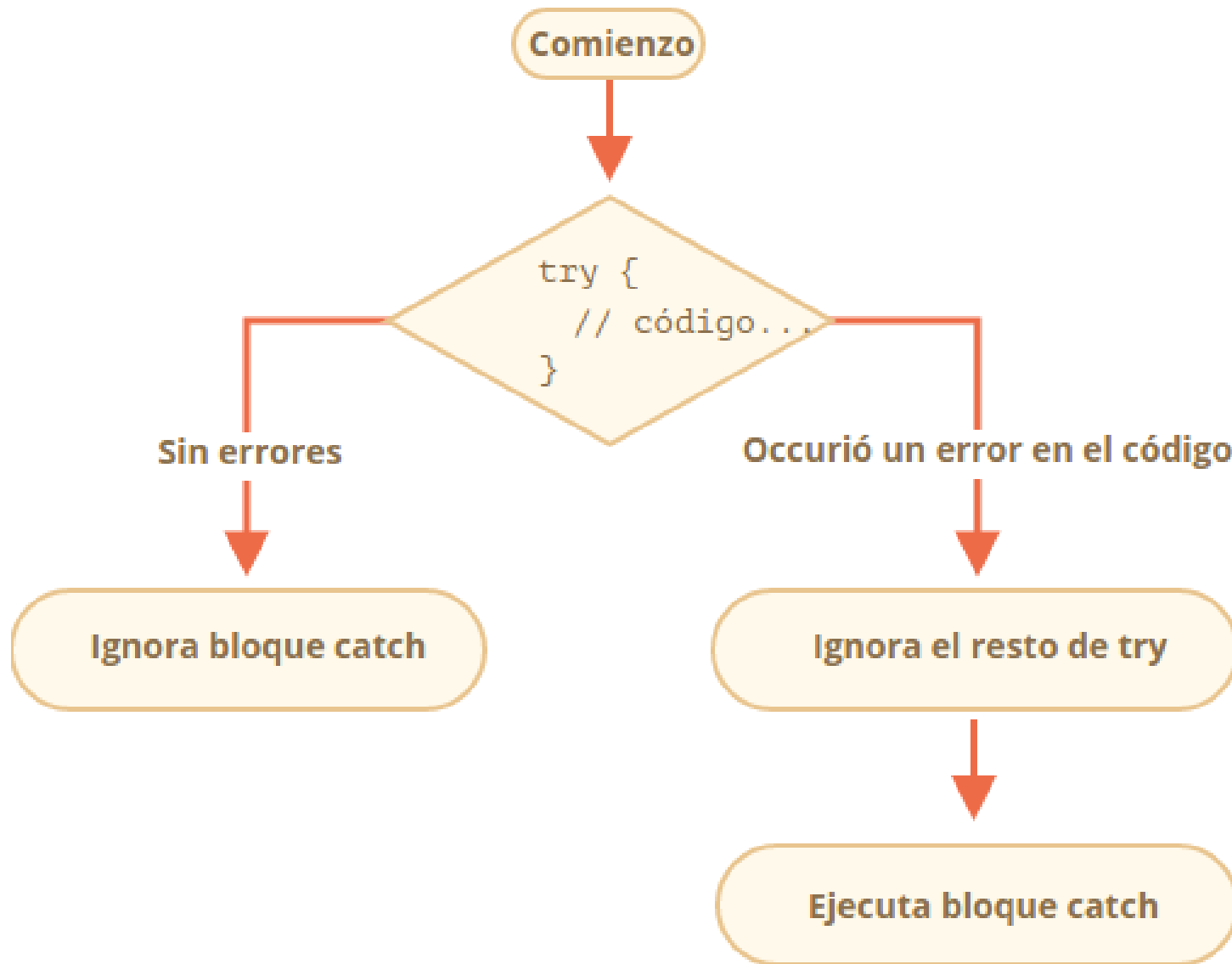
- `const MAX=2 // Correcto`  
`MAX=2 // NO Correcto`

- `Var x=2;`  
`Const y=3;`  
`y=x; // No Correcto`  
`y=4; // No Correcto`

- **Se debe asignar en la declaración:**

`const x; // Esta sentencia va a dar`

# try-catch-finally-throw



# try-catch-finally-throw

**Cuando se produce un error en Java Script se completa un objeto llamado err. err es un objeto que tiene 2 propiedades: message y name. name puede valer lo siguiente:**

- **RangeError:** Cuando se utiliza un número que está fuera de un rango permitido. Ejemplo:  
[https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_error\\_rangeerror](https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_rangeerror)
- **ReferenceError:** Cuando se utiliza una variable o función no declarada. Ejemplo:  
[https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_error\\_referenceerror](https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_referenceerror)
- **SyntaxError:** Estos errores normalmente generan una ruptura del hilo de ejecución.

# try-catch-finally-throw

**throw:** Constructores `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`.

Si con `throw` queremos lanzar un error personalizado podemos utilizar estos constructores. Se hará uso del constructor correspondiente al tipo de error producido. Por ejemplo si utilizamos `new TypeError(texto)` estamos creando un objeto error con propiedad `name` igual a `"TypeError"` y `message` igual al valor de la variable `texto`.

## Ejemplo:

```
try {
 if (x == "") throw "is empty";
 if (isNaN(x)) throw new TypeError("No
```