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) Funciones anidadas.
    3) Función flecha.
    4) Funciones predefinidas del lenguaje.
    5) Ámbito de las variables.
    6) Try-catch-finally-throw

1. Estructura de datos

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

Los arrays son unimensionales, 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 longitual

`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,arr){ //puedes no poner index ni arr

//item es el valor del elemento del array que se está valorando (las
modificaciones de esta variable no son visibles desde fuera de la función)

//index es la posición del array

//arr es el propio array, se utiliza para poder cambiar valores del array

text+="El elemento n° "+index+" tiene valor "+item+"<br>";

}

sistemaSolar.forEach(f1);


//text vale :

El elemento n° 0 vale Mercurio

El elemento n° 1 vale Venus

……
```

1.1. Recorrer un array II
Método `forEach()`

```
var text="";

function f1 (item,index,arr){

//item es el valor del elemento del array que se está valorando (las
modificaciones de esta variable no son visibles desde fuera de la función)

//index es la posición del array

//arr es el propio array, se utiliza para poder cambiar valores del array

arr[index]=item.toUpperCase()    //modifica el elemento en el array

// si se hubiese ejecutado item=item.toUpperCase() no se habría modificado el
array

}

sistemaSolar.forEach(f1);

//arr vale tras la ejecución de forEach

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 1° es el 0) hasta el fín:[“b”,”c”,”d”,”e”]

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

//devuelve desde el 4° por el final (el ultimo es el -1 no el 0), al
elemento 2° por el final [“b”,”c”]

array.slice(-4,-2)

//devuelve desde el 3° empezando por el principio, al elemento 2° 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,[“DWEC”,”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 ["DWEC", "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/oojs/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()`

3. 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();
```

3. 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.

3. 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.

3.1. 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.

3.1. 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

3.1. 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 literal definido por el usuario.
- Un objeto de una clase definida 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.

3.1. 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>
```

3.1. 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>
```

3.1. Funciones. Argumentos

Se puede invocar a una función sin pasar todos los argumentos definidos para la función.

Los parámetros no pasados existirán dentro de la función con valor "`undefined`"

```
function myFunction(x, y) {
  if (y==undefined) y=2;
  return x * y;
}
myFunction(4); //dev. 8. falta el 2° arg. que tomará el valor 2
```

3.1. Funciones. Objeto arguments

Las funciones de JavaScript cuentan con un objeto llamado `arguments`

Contiene un `array` con todos los argumentos pasados a la función al invocarla.

```
function suma() {

    // existe el array arguments
    // se puede utilizar arguments.length, arguments[1] …
    var res=0;
    for (i=0;i<arguments.length;i++){
        res+=arguments[i];
    }
    return res;

}

suma(4,5,10,22); //devuelve 41
```

3.2. Funciones anidadas I.

Podemos programar una función dentro de otra función.

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

3.2. 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.

3.2. Funciones anidadas III

Ejemplo:

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

3.3 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;
}

// Desglose de la función flecha
// Cualquiera de estas sintaxis son correctas

// 1. Elimina la palabra "function" y coloca la flecha entre el
argumento y las llaves de apertura.
ejemplo= (a) => {
  return a + 100;
}

// 2. Quita los corchetes del cuerpo y la palabra "return" — el
return está implícito. Esto es posible si solo hay una sentencia
dentro de la función.
ejemplo=(a) => a + 100;

// 3. Suprime los paréntesis de los argumentos (si solo hay 1 arg)
ejemplo=a => a + 100;
```

3.3 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;

// Practica
<script>
var hello;
var c1="hola";
var c2="de nuevo";
hello = (a,b) => a+” “+b;
document.getElementById("demo").innerHTML =
hello(c1,c2);
</script>
```

3.3 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 (sin argumentos)
a = 4;
b = 2;
Ejemplo= () => a + b + 100;

// Practica
var hello;
Var curso=”2DAWA”
hello = () => "Hola “+curso;
document.getElementById("demo").innerHTML =
hello();
```

3.3 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 suprimir las llaves.
ejemplo=(a, b) => {
  num = 42;
  return a + b + num;
}

// Practica
ejemplo=(a, b) => {
  num = 42;
  return a + b + num;
}
var n1=3;
var n2=4;
document.getElementById("demo").innerHTML = ejemplo(n1,n2);
```

3.3 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;

// Practica
<script>
sumar100 = a => a + 100;
var n1=3;
document.getElementById("demo").innerHTML = sumar100(n1);
</script>
```

3.3 Función flecha – forEach()

```
// Practica
var array=new Array("1","2");

// se elimina el nombre de la función y el =

array.forEach(elem=>{
      demo.innerHTML+=elem+index;
})


array.forEach((elem,index)=>{
      demo.innerHTML+=elem+index;
})
```

3.4. Funciones globales (funciones de objeto window)

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

`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.**

```
let a = 10;
let orden = “typeof a"; //Código javascript
let result = eval(orden);
demo.innerHTML+=result;
```

3.5 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, aún utilizando 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. Todas las instrucciones de tu script (incluidas las instrucciones que están dentro de las funciones), tendrán acceso directo al valor de esa variable.

En el momento que una página se cierra, todas las variables definidas en esa página se eliminarán de la memoria para siempre.

Si necesitas que el valor de una variable persista de una página a otra, tendrás que utilizar técnicas que te permitan almacenar esa variable (como las cookies, o en una base de datos, etc.).

La declaración de variables es opcional, pero se recomienda que la uses ya que así te protegerás de futuros cambios en las próximas versiones de JavaScript, y te funcionará en strict mode.

3.5. Ámbito de las variables II

Una variable local será definida dentro de una función. En este caso, sí **se requiere que se declare la variable,** ya que si no se declara 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.

Los parámetros de las funciones son variables locales.

3.5. Ámbito de las variables

- **En HTML una variable global solo es global para el objeto window en el que se ha declarado.**

```
let coche="fita"; // coche y window.coche es lo mismo
```

El tiempo de vida de una variable global es desde que se declara hasta que la venta se cierra.

- **Todas las funciones son métodos del objeto window:**

```
// parseInt() es lo mismo que window.parseInt()


function ordenar () {  //ordenar() es lo mismo que window.ordenar()
   // sentencias
}
//De hecho son variables de tipo Function var ordenar=function(){}
```

- **Una variable puede ser declarada después de utilizarse:**

```
x=3;
var x; // aquí x valdrá 3
```

3.5. Variables const

- **Para declarar variables cuyos valores no pueden cambiar se utiliza const. No es posible cambiar de valor, ni reasignarlo.**
    - ```
      const MAX=2  // Correcto
      MAX=3        // NO Correcto
      ```
    - ```
      var x=2;
      const y=3;
      y=x;  // No Correcto reasignarlos
      y=4;  // No Correcto cambiar de valor
      ```
- **Se debe asignar en la declaración:,**

```
const x; // Esta sentencia va a dar un error sintáctico (Syntac Error), ya que no se permite, declarar una variable const sin asignarle un valor
```

- **Redeclaración con const:**
- ```
  x=2;
  const x; // Esta sentencia va a dar un error sintáctico (Syntac Error), ya que no se permite
           // declarar una variable const sin asignarle un valor. Da este error antes de la redeclaración.
  ```
- ```
  Const x=2;
  Const x=3;  //NO es correcto. No podemos redeclarar si hemos declarado con const en el mismo ámbito
  ```
- ```
  Const x=2;
  { const x=3;}  // Correcto. Podemos redeclarar si hemos declarado con const también en distinto ámbito
  ```
- **En cuanto a ámbitos funcionan igual que let.**
  ```

3.5. Variables const

- **Se recomienda utilizar en objetos, funciones, array, y RegExp** (expresiones regulares). Ya que el valor de éstos es la dirección de memoria en la que está guardado su contenido, por lo que:
  - Se pueden cambiar propiedades o elementos del array.
  - No se pueden asignar a otra variable.

```
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");

cars = ["Toyota", "Volvo", "Audi"];    // ERROR


// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR

car2 = {type:"Volvo", model:"EX60", color:"red"};

car = car2 // ERROR
```

3.5. Variables de bloque

- **Se crean con let.** Un bloque de código son las sentencias incluidas entre llaves o entre paréntesis (en un for).
- **Las variables creadas con let en un bloque solo son visibles en ese bloque. Esto no ocurre si se declara con var**
-

```
{
      let x=2
}
// Aquí no se puede utilizar x
{
     var x=2
}
// Aquí se puede utilizar x
```

- **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 función con let su ámbito es local a la función igualmente.**

3.5. Redeclaraciones en distinto ámbito

- **Una variable (declarada con var o let o const) se puede redeclarar en otro ámbito con let o const, y si dentro del bloque la variable cambia, este cambio no se ve fuera del ámbito.**

```
var,let,const x=3
{
    let x=4-> ok, pero no cambia
    const x=4 -> ok pero no cambia el de fuera
}
```

Ejemplos:

-
```
let i=5;
for (let i=0;i<10;i++){
}
// Aquí i es igual a 5, no a 10
```

-
```
let i=5;
for (i=0;i<10;i++){
}
// Aquí i es igual a 10, no a 5 porque no se ha puesto let dentro del for
// luego i dentro del for está haciendo referencia a la variable i de fuera
```

3.5. Redeclaraciones en distinto ámbito

-
```
Var x=2;

{

      let x=5   // Aquí x es igual a 5

}

// Aquí x es igual a 2
```

- **Si se redeclara con var en un ambito una variable declarada con var fuera del ámbito, los cambios efectuados en la variable dentro del ámbito son visibles fuera.**

```
var x=3

{

    var x=4-> ok, pero x cambia fuera

}
```

- **No se puede redeclarar una variable con var si se ha declarado previamente con let o const en distinto ámbito.**

```
let,const x=3

{

    var x=4-> //ERROR

}
```

3.5. Redeclaraciones en el mismo ámbito

- **Solo se puede redeclarar una variable dentro del mismo ámbito, si declara y se redeclara con var.**
-
```
{ var x=2
    var x=3 } //es válido
```
-
```
{ var x=2
    let x=3 } // NO es válido//
```
```
  { let x=2
    var x=3 } // NO es válido
```
```
{
  let x=2
  let x=3
} //NO es válido
```
```
{
  const x=2
  let x=3
} //NO es válido
```

3.5. Variables de bloque

- **RECUERDA: Se puede redeclarar con let una variable ya declarada pero en otro ámbito, como ocurre en el bucle for.**

```
let x=2;

{

  let x=3

}

// x es igual a 2

var x=2;

{

let x=3   // Dentro x vale 3

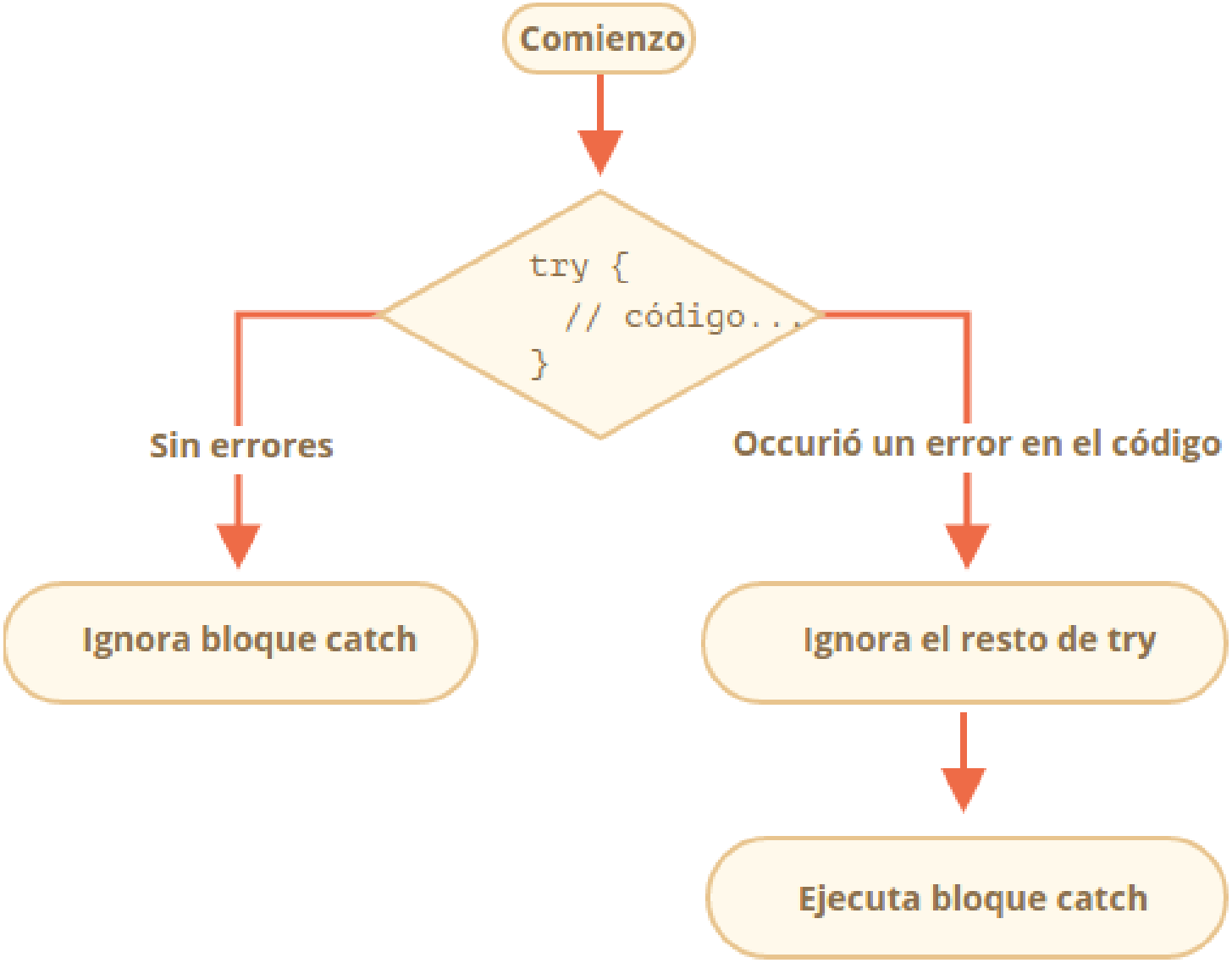
}

// Fuera x es igual a 2
```

3.6. try-catch-finally-throw

Un código Javascript se termina en caso de producirse un error durante su ejecución.

Para evitar la ruptura de un código se puede utilizar esta estructura de forma que el error es capturado y manejado evitando la ruptura del programa.



try-catch-finally-throw

Cuando se produce un error en Java Script se completa un objeto llamado `err` 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
- **ReferenceError:** Cuando se utiliza una variable o función no declarada. Ejemplo: https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_referenceerror
- **SyntaxError:** Errores de sintaxis, como falta el cierre de unas comillas, de un paréntesis, de unas llaves, etc. Estos errores normalmente generan una ruptura del hilo de ejecución. Pero dicho error puede ser capturado por el bloque `catch` si el código es ejecutado desde un método como `eval(), setInterval()`, etc. Ejemplo: https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_syntaxerror

  En este ejemplo prueba a poner en `eval()` este código que tiene un error de sintaxis `"var num=3;if (num==3{num=4}"`

- **TypeError:** Cuando se utiliza una variable que no es del tipo esperado**. Ejemplo:** https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_typeerror
- **URIError:** Cuando se utilizan caracteres no permitidos en una URI. Ejemplo:https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_urierror

Este error puede ser manejado de la siguiente forma. Sintaxis:

```
try {
  // Bloque de código a ejecutar y cuyos posibles errores queremos procesar
}
catch(err) {
  // Bloque de código que maneja el error producido
}
finally {
  // Bloque que se ejecuta siempre, ocurra un error o no
}
```

try-catch-finally-throw

`throw` → se utiliza para forzar un error dentro del bloque `try`. Este error será igualmente capturado por el bloque `catch`

Al generarse el error el bloque `try` deja de ejecutarse el resto de sentencias que hay a continuación dentro del `try` y el hilo de ejecución salta al bloque `catch`

El objeto error que se crea no es el objeto `err` por defecto con propiedades `name` y `message`.

En su lugar se crea un objeto específico con el contenido particularizado para dicho programa. Ese objeto error específico que se crea puede ser de cualquier tipo (`string`, `number`,`objeto`, etc.), y no tendrá las propiedades `name` y `message`, quedan como `undefined` ya que no se le da ningún valor.

Ejemplo de sintaxis: https://www.w3schools.com/js/tryit.asp?filename=tryjs_finally_error

En este ejemplo se ve cómo `throw` fuerza una serie de errores que el motor de Javascript del navegador no produciría por la ejecución del código, sino que se han producido de forma forzada a través del `throw`. Digamos que son errores creados de forma anticipada, controlada y organizada, antes de que el hilo de ejecución continúe y entonces provoque otros errores. Forma organizada quiere decir que se recogen todos los errores en una sola parte del código.

Los errores creados en todos los casos son `string ("……")` que son capturados por el bloque `catch` a través de la variable `err`. Los datos guardados en la variable `err` son utilizados dentro del bloque `catch`.

`throw` puede devover otro tipo de datos como un número o un objeto como en estos ejemplos:

```
throw 2    o   throw {"nombre":"Luis"}
```

Desde `catch` se utiizarían así: `err` o `err.nombre`

try-catch-finally-throw

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

Si con `throw` queremos lanzar errores `RangeError, ReferenceError, SyntaxError, TypeError,` etc. Pero con un mensaje de 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 has introducido un número");

//Estamos creando un objeto error con propiedad name "TypeError" y propiedad message "No has introducido un número

    if (num < 1 || num > 10) {
        throw new RangeError("Introduce un número entre 1 y 10.");
    }
}
catch(err) {
    message.innerHTML = "Input " + err.message;
}
```

try-catch-finally-throw

Ejercicio – notación exponencial

Crea una página web que tenga los siguientes elementos:

- **1 encabezado que ponga “Pasar a notación exponencial”**
- **1 input de tipo texto con una etiqueta llamada Número con decimales.**
- **1 input de tipo texto con una etiqueta llamada Cantidad de decimales.**
- **1 input de tipo botón llamado “Resolver”**
- **1 párrafo.**

Ejecuta la aplicación poniendo en “Cantidad de decimales” un 120 o un -2.

¿Qué ocurre?

Utiliza try-catch para evitar que el programa se rompa.

try-catch-finally-throw

Ejercicio – notación exponencial - continuación

Continua con el ejercicio anterior, pero utilizando try-catch-finally-throw y también los constructores de throw crea la función que se va a ejecutar al presionar el botón “Resolver”. De forma que:

1. **Si el usuario no completa alguno de los dos inputs salga en el párrafo un mensaje de error y el color de fondo de los inputs tipo texto cambia a rojo.**
2. **Si el usuario no introduce un número en alguno de los dos inputs salga en el párrafo el mensaje de error y el color de fondo de los inputs tipo texto cambia a rojo.**
3. **Si el usuario introduce una cantidad de decimales negativo salga en el párrafo un mensaje de error y el color de fondo de los inputs tipo texto cambia a rojo.**
4. **Si el usuario introduce una cantidad de decimales por encima de 100 salga en el párrafo un mensaje de error y el color de fondo de los inputs cambia a rojo.**
5. **Si el usuario introduce todos los datos correctamente, salga en el párrafo un mensaje indicando el número en notación exponencial y el color de fondo de los inputs cambia a blanco.**
6. **Siempre se ha de borrar el contenido de los inputs.**

Realiza estos 6 puntos SIN UTILIZAR TRY-CATCH-FINALLY-THROW. Verás que el código queda menos claro.

try-catch-finally-throw

Ejercicio - introducción de email

Crea una página web que tenga los siguientes elementos:

- **1 input de tipo texto con una etiqueta llamada email**
- **1 input de tipo texto con una etiqueta llamada Confirmación email**
- **1 input de tipo botón llamado “Comprobar”**
- **1 párrafo.**

Utilizando try-catch-finally-throw y también los constructores de throw crea la función que se va a ejecutar al presionar el botón “Comprobar”. De forma que:

1. **Si el usuario no completa los inputs salga en el párrafo un mensaje de error y el color de fondo de los inputs tipo texto cambia a rojo.**
2. **Si el usuario no introduce dos emails iguales salga en el párrafo un mensaje de error y el color de fondo de los inputs tipo texto cambia a rojo.**
3. **Si el usuario no introduce un email correcto salga en el párrafo un mensaje de error y el color de fondo de los inputs cambia a rojo.. Un email correcto es:**
   a) **Debe tener el carácter @**
   b) **Después del carácter @ debe haber al menos un carácter “.”**
4. **Si el usuario introduce todos los datos correctamente, salga en el párrafo un mensaje indicando la dirección email introducida y que es correcta y el color de fondo de los inputs cambia a blanco.**
5. **Siempre se ha de borrar el contenido de los inputs.**