UT 9 Control de errores y programación Asíncrona.

Introducción Errores del usuario.

Es una labor difícil, en ocasiones los errores no se detectan ya que suele ser una habilidad la de encontrar los errores, que se adquiere con los años, una de las etapas en el proceso de desarrollo de aplicaciones es la prueba, donde buscamos encontrar posibles errores.

Los errores pueden ser de distintos tipos:

- **Errores de sintaxis**. Son fallos mientras que se teclea en expresiones, falta de paréntesis, llaves, … Normalmente son marcados por el editor con subrayando en rojo.
  - **Errores detectados en tiempo de escritura**. Son los errores evidentes que nos suele detectar el editor con un subrayado rojo.
  - **Errores de ejecución.** Son aquellos que solo se detectan al ejecutar la aplicación. Por ejemplo usar variables no declaradas, …
- **Errores lógicos.** La aplicación no hace lo que debe. No hay herramientas para detectarlos, en ocasiones lo detectan los usuarios.
- **Errores del sistema.** Son causados por eventos externos, como fallos de conexión en la red, ...
- **Errores del usuario.** Se producen cuando el usuario realiza alguna acción que no ha sido contemplada por el programador. Como introducir texto en un campo numérico, dejar alguna opción sin activar, ...

Errores, Excepciones y avisos.

Errores. Fallo en un programa que provoca que se detenga.

Excepción. Error en un programa que puede ser controlado. Mediante la gestión de errores podemos reconducir, para solventarlo y que siga funcionando correctamente la aplicación.

Aviso o warning. Error leve, no impide la ejecución de un programa, se avisa al programador para que sea consciente, ya que pueden ser una forma de detectar posibles errores más graves.

Javascript Estricto.

Javascript se creo para ser rápido, dinámico y su sintaxis es poco rígido, reduciendo el esfuerzo del desarrollo de aplicaciones.

La directiva ‘**use strict**’ no supone una instrucción de código, indica el modo en que el navegador debe ejecutar el código. Dos modos de ejecución JavaScript: el ‘’**normal mode**’’, que es el que hemos estudiado hasta ahora, y el ‘’**strict mode**’’

Funciona en modo normal pero no admitida en strict mode	Ejemplo
Uso de variables no declaradas	Edad=19;
Definir una propiedad varias veces	Let lista={apellido:’Perez’,apellido=’Diaz’}
Nombres de parámetros duplicados	function ejemplo(palabra,palabra) { };
Usar eval como nombre de variable	eval=’mayor’;
Otras	Hay más restricciones que impone el strict mode, uso de sintaxis o instrucciones poco adecuadas, prácticas inseguras y mejora de la seguridad para el usuario.

Ejemplo

```
<script>
    'use strict';
    x=9;
    console.log(x);
</script>
```

```
top
Filter
Uncaught ReferenceError: x is not defined
    at UsoEstrictoI.html:11
Live reload enabled.
>
```

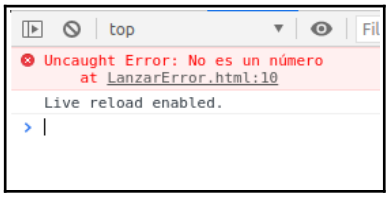
```
<script>
    escribe("el texto siguiente");

    function escribe(t){
        'use strict';
        inicio="Empeza";
        console.log(inicio, " ",t);
    }
</script>
```

```
top
Filter
Uncaught ReferenceError: inicio is not defined
    at escribe (UsoEstrictoII.html:14)
    at UsoEstrictoII.html:10
Live reload enabled.
>
```

Crear y lanzar errores propios

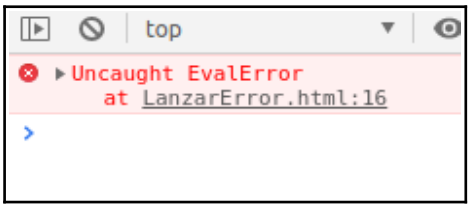
Los errores los crea el intérprete de Javascript cuando aparecen, pero también podemos nosotros lanzar errores según nuestras necesidades. Podemos crear errores genéricos como los del ejemplo, con el objeto error o utilizar alguno de los más específicos.

```
<script>
    let miError=new Error("No es un número");
    let x='a';
    if (isNaN(x)){
        throw miError;
    }
</script>
```



Tipo de Error	Uso
EvalError	Error al intentar usa la función eval()
RangeError	Error número o de rango de valores incorrectos
RefenceError	Referenciamos un objeto no valido.
TypeError	Error por mal uso del tipo de dato.
URLError	Error al codificar o decodificar una URL

```
let miError2=new EvalError();
throw miError2;
```



GESTIÓN DE ERRORES

Para la gestión de errores utilizamos en Javascript la estructura try, catch.

La sentencia **try** consiste en un bloque **try** que contiene una o más sentencias. Las llaves **{}** se deben utilizar **siempre**, incluso para una bloques de una sola sentencia. **Al menos** un bloque **catch o** un bloque **finally** debe estar presente. Esto nos da tres formas posibles para la sentencia try:

1. `try...catch`
2. `try...finally`
3. `try...catch...finally`

Un bloque **catch** contiene sentencias que especifican que hacer si una excepción es lanzada en el bloque **try**.

Si cualquier sentencia dentro del bloque **try** (o en una función llamada desde dentro del bloque **try**) lanza una excepción, el control cambia inmediatamente al bloque **catch** .

Si no se lanza ninguna excepción en el bloque **try**, el bloque **catch** se omite.

La bloque **finally** se ejecuta después del bloque **try** y el/los bloque(s) **catch** hayan finalizado su ejecución. Éste bloque **siempre se ejecuta**, independientemente de si una excepción fue lanzada o capturada.

`Ejemplo catch incondicional`

Cuando solo se utiliza un bloque catch, el bloque catch es ejecutado cuando cualquier excepción es lanzada.

Por ejemplo, cuando la excepción ocurre en el siguiente código, el control se transfiere a la cláusula catch.

```
try {
   throw "myException"; // genera una excepción
}
catch (e) {
   // sentencias para manejar cualquier excepción
   logMyErrors(e); // pasa el objeto de la excepción al manejador de errores
}
```

El bloque catch especifíca un identificador (e en el ejemplo anterior) que contiene el valor de la excepción. Este valor está solo disponible en el scope de el bloque catch.

Bloques catch condicionales

También se pueden crear "bloques catch condicionales", combinando bloques try...catch con estructuras if...else if...else como estas:

```
try {
       myroutine();  // puede lanzar tres tipos de excepciones
} catch (e) {
   if (e instanceof TypeError) {
      // sentencias para manejar excepciones TypeError
```

```
    } else if (e instanceof RangeError) {
        // sentencias para manejar excepciones RangeError
    } else if (e instanceof EvalError) {
        // sentencias para manejar excepciones EvalError
    } else {
      // sentencias para manejar cualquier excepción no especificada
      logMyErrors(e); // pasa el objeto de la excepción al manejador
de errores
}
```

Creación de módulos

Un módulo no es mas que un fichero js, con funciones y declaraciones, donde alguna de ellas serán accesibles y otras no desde fuera del fichero.

La declaración **export** se utiliza al crear módulos de JavaScript para exportar funciones, objetos o tipos de dato primitivos del módulo para que puedan ser utilizados por otros programas con la sentencia **import**.

```
export const PI_Al_CUADRADO=Math.PI**2

export function areaCirculo(radio){
    return Math.PI*radio**2
}

export function areaCuadrado(lado){
    return ladoAlCuadrado(lado)
}

export function areaHexagoR(lado){
    return 6*(ladoAlCuadrado(lado)*Math.sqrt(3))/4
}

function ladoAlCuadrado(lado){
    return lado*lado
}
```

Otra forma de indicar que exportamos y es usar exports con nombres.

```
21  const PI=Math.PI**2;
22
23  function areaCirculo(radio){
24      return Math.PI*radio**2
25  }
26
27  function areaCuadrado(lado){
28      return ladoAlCuadrado(lado)
29  }
30
31  function areaHexagoR(lado){
32      return 6*(ladoAlCuadrado(lado)*Math.sqrt(3))/4
33  }
34
35  function ladoAlCuadrado(lado){
36      return lado*lado
37  }
38
39  export{
40      PI,
41      areaCirculo,
42      areaHexagoR
43  }
```

Importación de funciones y otros elementos de un módulo

La sentencia **import** se usa para importar funciones que han sido exportadas desde un módulo externo.

Importa un solo miembro de un módulo.

```
<script type="module">
    import {areaCirculo } from "./moduloAreas.js";

    console.log(areaCirculo(5));
</script>
```

Importa múltiples miembros de un módulo.

```
<script type="module">
    import {areaCirculo, PI } from "./moduloAreas.js";

    console.log(areaCirculo(5)," y pi al cuadro es ",PI);
</script>
```

Renombra múltiples miembros durante la importación

```
<script type="module">
    import {areaCirculo as aC, PI as pi} from "./moduloAreas.js

    console.log(aC(5)," y pi al cuadro es ",pi);
</script>
```

Importa el contenido de todo un módulo.

```
<script type="module">
    import * as areas from "./moduloAreas.js";

    console.log(areas.areaCirculo(5)," y pi al cuadro ",areas.PI);
</script>
```