ERRORES, EXCEPCIONES Y PROMESAS

Excepciones y sentencia **throw**

- Una excepción es una señal que interrumpe un programa.
  - La señal (excepción) se interrumpe con la sentencia **throw <msg>**
- La excepción lleva un valor asociado que se utiliza para identificarla. Ej:

```
throw 'Abort Execution'
```

- Code Example:

```
1   console.log('this msg will be shown');
2
3   throw 'abort execution';
4
5   console.log("this msg won't be shown");
```

```
rglep@RGL-LAP /cygdrive/c/Users/rglep/OneDrive/Documents/Curso CIFO-La Violeta/Mean Stack 2019/MS-1/Ejercicios/excepciones
$ node exception
this msg will be shown

c:\Users\rglep\OneDrive\Documents\Curso CIFO-La Violeta\Mean Stack 2019\MS-1\Ejercicios\excepciones\exception.js:3
throw 'abort execution';
^
abort execution
```

Errores

- Los **errores son excepciones** con un valor de la clase predefinida **Error**.
- También se lanzan con la sentencia **throw**:

```
throw  new Error (‘Abort Execution’)
```

- Ejemplo:

```
1    console.log('this msg will be shown');
2
3    throw new Error('abort execution');
4
5    console.log("this msg won't be shown");
6
```

```
$ node exception
this msg will be shown
c:\Users\rglep\OneDrive\Documents\Curso CIFO-La Violeta\Mean Stack 2019\MS-1\Ejercicios\excepc
throw new Error('abort execution');
^

Error: abort execution
    at Object.<anonymous> (c:\Users\rglep\OneDrive\Documents\Curso CIFO-La Violeta\Mean Stack
xception.js:3:7)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:663:10)
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
    at Function.Module._load (module.js:497:3)
    at Function.Module.runMain (module.js:693:10)
    at startup (bootstrap_node.js:191:16)
    at bootstrap_node.js:612:3
```

Errores de ejecución

- El intérprete de JavaScript analiza las instrucciones del programa en tiempo de ejecución:
  - Si encuentra un error lanza una excepción con un valor de la clase **Error**:
  - Los errores que provocan excepciones son, por ejemplo:
    - Funciones no definidas,
    - Métodos no definidos,
    - Variables no definidas,
    - Detectar errores de sintaxis,
    - …
- En el ejemplo se invoca una función no definida que lanza un **ReferenceError** (clase derivada de Error). Cada tipo de error tiene asociada una clase predefinida: ReferenceError, SyntaxError, RangeError,….

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

```
1   console.log('this msg will be shown');
2
3   undefinedFunction();
4
5   console.log("this msg won't be shown");
6
```

```
$ node exception
this msg will be shown
c:\Users\rglep\OneDrive\Documents\Curso CIFO-La V
undefinedFunction();
^

ReferenceError: undefinedFunction is not defined
    at Object.<anonymous> (c:\Users\rglep\OneDriv
xception.js:3:1)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:6
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
```

Sentencias try…catch…finally

- Las excepciones y los errores **interrumpen la ejecución** de un programa salvo si son capturadas dentro de un bloque **try** de la sentencia try…catch…finally:
  - **catch** captura excepciones o errores ocurridos dentro de **try**

    La ejecución continua en el bloque catch.
  - El bloque **finally** se ejecuta siempre, haya o no excepciones o errores.

- try…catch captura cualquier error:
  - Excepciones o errores del programa
  - Errores lanzados por el intérprete

```
rglep@RGL-LAP /cygdrive/c/Users/rglep/OneDrive/Documents/Curso CIFO
$ node exception
this msg will be shown
ERROR CAPTURED =>ReferenceError: undefinedFunction is not defined
this msg ALSO will be shown
```

```
1  try {
2
3    console.log('this msg will be shown');
4
5    undefinedFunction();
6
7    console.log("this msg won't be shown");
8    }
9  catch(error){
10     console.log('ERROR CAPTURED =>'+ error);
11   }
12
13   console.log('this msg ALSO will be shown')
14
```

Pongámoslo en práctica

1ª Iteración

- Vamos a buscar una persona en un array de personas:

```
let people =    [{name: 'Sophie', age: 25},
                 {name: 'Christian', age: 50},
                 {name: 'Anna', age: 99}]
```

- Si la persona existe devuelve el nombre de la persona y sino, devuelve **undefined**. La búsqueda de una propiedad de un objeto undefined devuelve un error, por lo que lo controlaremos con una sentencia try…catch.

- Usaremos la función prompt del objeto window para solicitar el valor.

2ª Iteración

- Lanzaremos nosotros la excepción en el bloque try para controlar el error.

Promesas

- ES6 incluye una nueva clase llamada **Promise** (Promesa)
  - Una promesa es una tarea que promete devolver un valor en un futuro

- La promesa tiene tres estados para gestionar este proceso:
  - *Pending:* antes de ejecutar la promesa,
  - *Fullfilled:* la tarea tiene **éxito** y devuelve el valor prometido,
  - *Error:* la tarea **falla** y genera un código de rechazo (en lugar del valor prometido)

- Las promesas simplifican la programación asíncrona
  - Permiten componer funciones asíncronas como si fueran síncronas
    - Permiten separar claramente el código de **éxito** del código de **atención a errores**
  - Conserva la eficiencia de los 'callbacks' asíncronos

Promesas: constructor, resolve y reject

- La promesa se construye invocando el constructor de la clase **Promise**, al que se le pasa una función ***ejecutora***.
  - El ejecutor es un *callback* al que se le pasan dos parámetros: **resolve** y **reject.**
  - Si sólo queremos tratar el caso de éxito podemos crear la promesa a partir del método estático: **Promise.resolve(<value>)**

- **resolve(<value>)**
  - Al invocar una función **resolve** la promesa finaliza con éxito y genera el valor prometido.
  - **value** es el valor generado por la promesa. Puede ser de cualquier tipo, incluso otra promesa.
- **reject(<reason>)**
  - Al invocar a la función reject, la promesa se rechaza.
  - **reason** es la causa del rechazo de la promesa, normalmente describe la causa del fallo.

Métodos **then y catch**

- Todas las promesas tienen un método para acceder al resultado devuelto por la promesa.
  - El método **then** acepta dos *callbacks:* el primero se invoca cuando la promesa se resolvió con éxito **(resolve).** El segundo para el caso de rechazo **(reject).**
- En el caso de rechazo se puede utilizar el método **catch**() que equivale a ejecutar el segundo callback del método then.
- Ambos métodos, a su vez, devuelven una promesa que puede ser evaluada.
- Para promesas encadenadas con un método catch para todas las promesas es suficiente.

Pongámoslo en práctica

Ejemplo de manejo de errores con promesas.

```
function imgLoad(url) {
 return new Promise(function(resolve, reject) {
        let request = new XMLHttpRequest();
        request.open('GET', url);
        request.responseType = 'blob’;
        request.onload = function() {
                if (request.status === 200) {
                resolve(request.response);
                } else {
                reject(Error('Image didn\'t load successfully; error code:' +
request.statusText)); } };
        request.onerror = function() {
                reject(Error('There was a network error.')); };
                request.send(); }); }

imgLoad('example.com/imagen').then( imagen =>console.log(imagen),
error=>{console.log(error)} )
```