06. ERROR HANDLING JAVASCRIPT

IES ESTACIÓ CURS 2021- 2022

Index

1. What is an error in programming?

- There are situations where we may want to stop the program or inform the user if something bad happens.

- For example:
  - the program tried to open a non-existent file.
  - the network connection is broken.
  - the user entered invalid input.

- In all these cases we as programmers, create errors, or we let the programming engine create some for us.

- After creating the error we can inform the user with a message, or we can stop the execution.

2. What is an error in JavaScript?

- An error in JavaScript is an object, which is later thrown to halt the program.

- To create a new error in JavaScript we call the appropriate constructor function. For example, to create a new, generic error we can do:

```
const err = new Error ("Something bad happened!!");
```

- When creating an error object it's also possible to omit the new keyword:

```
const er = Error ("Ops! Something went wrong!");
```

2. What is an error in JavaScript?

- Once created, the error object presents three properties:
  - **message**: a string with the error message.
  - **name**: the error's type.
  - **stack**: a stack trace of functions execution.
- For example, if we create a new **TypeError** object with the appropriate message, the message will carry the actual error string, while name will be "TypeError":

```
const wrongType = TypeError ("Wrong type given, expected number");

console.log(wrongType.message); //"Wrong type given, expected number"
console.log(wrongType.name); //"TypeError"
```

3. Many types of errors in JavaScript.

- There are many types of errors in JavaScript, namely:
  - Error
  - EvalError
  - InternalError
  - RangeError
  - ReferenceError
  - SyntaxError
  - TypeError
  - URIError

- Remember, all these error types are actual constructor functions meant to return a new error object.

3. Many types of errors in JavaScript.

- In your code you'll mostly use **Error** and **TypeError**, two of the most common types, to create your own error object.

- Most of the times, the majority of errors will come directly from the JavaScript engine, like **InternalError** or **SyntaxError**.

- An example of **TypeError** occurs when you try to reassign const:

```
const name = "Jules";
name = "Caty";

// TypeError: Assignment to constant variable.
```

- An example of **SyntaxError** is when you misspell language keywords:

```
va x = '33';

//SyntaxError: Unexpected identifier
```

3. Many types of errors in JavaScript.

- Or when you use reserved keywords in wrong places, like await outside of an async function:

```
function wrong(){
    await 99;
}

wrong();

// SyntaxError: await is only valid in async function
```

- Another example of **TypeError** occurs when we select non-existent HTML elements in the page:

```
// Uncaught TypeError: button is null
```

- **DOMException** is a family of errors related to Web APIs. They are thrown when we do silly things in the browser, like:

```
document.body.appendChild(document.cloneNode(true));

// Uncaught DOMException: Node.appendChild: May not add a Document as a child
```

4. What is an exception?.

- Most developers think that error and exceptions are the same thing. In reality, an error object becomes an exception only when it's thrown.
- To throw an exception in JavaScript we use throw, followed by the error object:

```
const wrongType = TypeError("Wrong type given, expected number");

throw wrongType;
```

- The short form is more common:

```
throw TypeError("Wrong type given, expected number");
```

- Or:

```
throw new TypeError("Wrong type given, expected number");
```

4. What is an exception?.

- It's unlikely to throw exceptions outside of a function or a conditional block. Instead, consider the following example:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}
```

- Here we check if the function argument is a string. If it's not we throw an exception.
- Technically, you could throw anything in JavaScript, not only error objects:

```
throw Symbol();
throw 33;
throw "Error!";
throw null;
```

- However, it's better to avoid these things: always throw proper error objects, not primitives.
- By doing so you keep error handling consistent through the codebase.

5. What happens when we throw an exception?.

- Exceptions are like an elevator going up: once you throw one, it bubbles up in the program stack, unless it is caught somewhere.
- Consider the following code:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

toUppercase(4);
```

5. What happens when we throw an exception?.

- If you run this code in a browser, the program stops and reports the error:

```
Uncaught TypeError: Wrong type given, expected a string
    toUppercase http://localhost:5000/index.js:3
    <anonymous> http://localhost:5000/index.js:9
```

- In addition, you can see the exact line where the error happened.
- This report is a stack trace, and it's helpful for tracking down problems in your code.
- The stack trace goes from bottom to top. So here:

```
   toUppercase http://localhost:5000/index.js:3
   <anonymous> http://localhost:5000/index.js:9
```

5. What happens when we throw an exception?.

- We can say:
  - something in the program at line 9 called toUppercase
  - toUppercase blew up at line 3
- In addition to seeing this stack trace in the browser's console, you can access it on the stack property of the error object.
- If the exception is uncaught, that is, nothing is done by the programmer to catch it, the program will crash.
- When, and where you catch an exception in your code depends on the specific use case.
- For example you may want to propagate an exception up in the stack to crash the program altogether. This could happen for fatal errors, when it's safer to stop the program rather than working with invalid data.
- Having introduced the basics let's now turn our attention to error and exception handling in both synchronous and asynchronous JavaScript code.

6. Synchronous error handling.

- Synchronous code is most of the times straightforward, and so its error handling.
- 6.1 Error handling for regular functions.
  - Synchronous code is executed in the same order in which is written. Let's take again the previous example:

```
function toUppercase(string) {
  if (typeof string !== "string") {
    throw TypeError("Wrong type given, expected a string");
  }

  return string.toUpperCase();
}

toUppercase(4);
```

  - Here the engine calls and executes toUppercase. All happens synchronously. To catch an exception originating by such synchronous function we can use try/catch/finally:

```
try {
  toUppercase(4);
} catch (error) {
  console.error(error.message);
  // or log remotely
} finally {
  // clean up
}
```

6. Synchronous error handling.

- Synchronous code is most of the times straightforward, and so its error handling.
- 6.1 Error handling for regular functions.
  - Usually, try deals with the happy path, or with the function call that could potentially throw.
  - catch instead, captures the actual exception. It receives the error object, which we can inspect (and send remotely to some logger in production).
  - The finally statement on the other hand runs regardless of the function's outcome: whether it failed or succeeded, any code inside finally will run.
  - Remember: try/catch/finally is a synchronous construct: it has now way to catch exceptions coming from asynchronous code.

7. Asynchronous error handling.

- JavaScript is synchronous by nature, being a single-threaded language.
- Host environments like browsers engines augment JavaScript with a number of Web API for interacting with external systems, and for dealing with I/O bound operations.
- Examples of asynchronicity in the browser are timeouts, events, Promise.
- Error handling in the asynchronous world is distinct from its synchronous counterpart.
- Let's see some examples.

7. Asynchronous error handling.

- 7.1 Error handling for timers.
  - In the beginning of your explorations with JavaScript, after learning about try/catch/finally, you might be tempted to put it around any block of code.
  - Consider the following snippet:

```
function failAfterOneSecond() {
  setTimeout(() => {
    throw Error("Something went wrong!");
  }, 1000);
}
```

  - This function throws after roughly 1 second. What's the right way to handle this exception?
  - The following example does not work:

```
function failAfterOneSecond() {
  setTimeout(() => {
    throw Error("Something went wrong!");
  }, 1000);
}

try {
  failAfterOneSecond();
} catch (error) {
  console.error(error.message);
}
```

7. Asynchronous error handling.

- 7.1 Error handling for timers.
  - As we said, try/catch is synchronous. On the other hand we have setTimeout, a browser API for timers.
  - By the time the callback passed to setTimeout runs, our try/catch is long gone. The program will crash because we failed to capture the exception.
  - They travel on two different tracks:

```
Track A: --> try/catch
Track B: --> setTimeout --> callback --> throw
```

  - If we don't want to crash the program, to handle the error correctly we must move try/catch inside the callback for setTimeout.
  - But, this approach doesn't make much sense most of the times. As we'll see later, asynchronous error handling with Promises provides a better ergonomic.

7. Asynchronous error handling.

- 7.1 Error handling for events.
  - HTML nodes in the Document Object Model are connected to EventTarget, the common ancestor for any event emitter in the browser.
  - That means we can listen for events on any HTML element in the page.
  - The error handling mechanics for DOM events follows the same scheme of any asynchronous Web API.
  - Consider the following example:

```
const button = document.querySelector("button");

button.addEventListener("click", function() {
  throw Error("Can't touch this button!");
});
```

7. Asynchronous error handling.

- 7.1 Error handling for events.
  - Here we throw an exception as soon as the button is clicked. How do we catch it? This pattern does not work, and won't prevent the program from crashing:

```
const button = document.querySelector("button");

try {
  button.addEventListener("click", function() {
    throw Error("Can't touch this button!");
  });
} catch (error) {
  console.error(error.message);
}
```

7. Asynchronous error handling.

- 7.1 Error handling for events.
  - As with the previous example with setTimeout, any callback passed to addEventListener is executed asynchronously:

```
Track A: --> try/catch
Track B: --> addEventListener --> callback --> throw
```

  - If we don't want to crash the program, to handle the error correctly we must move try/catch inside the callback for addEventListener.
  - But again, there's little of no value in doing this.
  - As with setTimeout, exception thrown by an asynchronous code path are un-catchable from the outside, and will crash your program.
  - When we see Promises and async/await can ease error handling for asynchronous code.