Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Asynchrony and APIs (/intermediate-javascript/asynchrony-and-apis) / JavaScript Exception Handling with try...catch

Text

In this lesson, we're going to familiarize ourselves with a JavaScript exception handling tool called `try...catch`. This tool can be used for any JavaScript, as well as code that makes API calls. In a future lesson, we'll use a `try...catch` with an API call, but in this lesson, we'll use it with regular, synchronous JavaScript. So, take the opportunity to absorb the concepts covered here with the understanding that they will be incorporated later in this section.

Error Handling with `try...catch`

In an ideal world, all our code would function perfectly and never have errors. But in the real world, errors are inevitable. By this point, you've learned how to debug applications with breakpoints and the DevTools console. You've also learned to lint and to continuously test your code with Jest. Let's take this knowledge one step further and learn about exception handling.

First, what is an exception? An **exception** is an unusual problem that arises in your code. An exception should be just that: *exceptional*. What does "exceptional" mean? Exceptions should handle *unexpected* errors in our code, not *anticipated* errors. When a user enters their password incorrectly, that's an *anticipated* error. Users often make mistakes, so we shouldn't throw exceptions when they do.

However, let's say we have a complex application that handles credit card payments. What would happen if we had a `payBalance()` function that accidentally charged our customers twice? That would be a serious and unexpected error.

Similarly, when we make an API call, we expect to get a `200 OK` as a response. However, if we were to get a `400` or `500` level status code (error codes), that would be an exceptional error.

Programmers use **exception handling** to deal with serious and unexpected anomalies in their code. Exception handling is a feature of programming languages in general, not just JavaScript.

In JavaScript, we can handle an exception with a `try...catch` block, which looks like this:

```
try {
  // Code to try goes here.
} catch {
  // Handle or log any raised errors.
}
```

We can wrap any code inside the `try` block. Then, if that code has errors, control will shift to the `catch` block, where we can write code to handle these errors.

What do we mean by control? Well, **control flow** is a term used to explain the order a sequence of code will be evaluated. Conditionals change the flow of control in a block of code depending on whether the conditional evaluates to true or false. Likewise, a `try...catch` block can change control flow if an error is thrown, moving the control into the `catch` block.

We can't use a `try` block by itself; doing so will throw an error. `try` blocks must always be accompanied by either `catch`, `finally`, or both. We won't cover `finally` in depth, other than the fact that a `finally` block always runs regardless of whether the `try` block has errors that are caught. `finally` blocks are often used for cleanup or freeing up resources.

A Demonstration

Let's create a very basic application to demonstrate how to use `try...catch` blocks. The application asks a user to input a number. If the number is negative, the application will throw and catch an error.

Before we start, **it's important to note that this is not a situation where we'd use exception handling**. After all, we expect users to make mistakes. However, we can use this example to show how exception handling works.

The root directory of our application will have two files: `try.html` and `try.js`. Note that we aren't using a development environment for this example — we don't need webpack to demonstrate how `try...catch` blocks work. We will add `try...catch` blocks in a development environment in a future lesson.

Here's the HTML:

try.html

```
<html lang="en-US">
<head>
  <script type="text/javascript" src="try.js"></script>
  <title>Enter a positive number</title>
</head>
<body>
  <div>
    <h1>Please enter a whole number above 0</h1>
    <form>
      <label for="number">Enter your number:</label>
      <input id="number" name="number" type="text">
      <button type="submit" id="submittedNumber">Is your nu
mber valid?</button>
    </form>
    <div id="displayNumber"></div>
  </div>
</body>
</html>
```

Now let's take a look at the JavaScript code:

try.js

```
// User Interface Logic
function checkNumber(number) {
  if (isNaN(number) || number < 0) {
    throw "Not a valid number!";
  } else {
    document.querySelector('#displayNumber').innerText = "T
his number is valid. You may continue.";
  }
}

window.addEventListener("load", function() {
  document.querySelector('#submittedNumber').addEventListen
er("submit", function(event) {
    event.preventDefault();

    const inputtedNumber = parseInt(document.querySelector
('#number').value);
    document.querySelector('#number').value = null;

    try {
      checkNumber(inputtedNumber);
    } catch(error) {
      console.error(`Red alert! We have an error: ${error.m
essage}`);
    }
  });
});
```

We'll skip the familiar event handling code and jump right into the new concepts. First, we have a `checkNumber()` function which will check to see if the number is `NaN` or below 0. If it is, it will `throw` an exception. Here's ours:

```
if (isNaN(number) || number < 0) {
  throw "Not a valid number!";
}
```

The `throw` statement is very similar to `return` in that it ends the current function in which it is called. So, if we included more code below our `throw` statement, it would not run:

```
if (isNaN(number) || number < 0) {
  throw "Not a valid number!";
  // this console.log is unreachable
  console.log("Hello!");
}
```

However, `throw` is different from the `return` keyword in that control automatically switches to a `catch` block, if there is one. If the program can't find a `catch`, the program will simply terminate. So, if we intend to use the `throw` keyword effectively, we should always use it with `try...catch`. That's just what we do:

```
try {
  checkNumber(inputtedNumber);
} catch(error) {
  console.error(`Red alert! We have an error: ${error.messa
ge}`);
}
```

That's the basics of a `try...catch` block. However, there's a few ways we can improve our code: we can turn our `checkNumber()` function into a business logic function, and we can incorporate the JavaScript `Error` object. In the process, we'll see another way that we can use a `try...catch`, and learn about a new operator.

Let's do that next!

`try...catch` with Separated Logic

For this next configuration, we'll only be updating our code in `try.js`. Here's what the new code looks like:

try.js

```
// Business Logic
function checkNumber(number) {
  if (isNaN(number) || number < 0) {
    return new Error("Not a valid number!");
  } else {
    return true;
  }
}

// User Interface Logic
window.addEventListener("load", function() {
  document.querySelector('#submittedNumber').addEventListener("submit", function(event) {
    event.preventDefault();

    const inputtedNumber = parseInt(document.querySelector('#number').value);
    document.querySelector('#number').value = null;

    try {
      const isNumberValid = checkNumber(inputtedNumber);
      if (isNumberValid instanceof Error) {
        console.error(isNumberValid.message);
        throw RangeError("Not a valid number!");
      } else {
        console.log("Try was successful, so no need to catch!");
        document.querySelector('#displayNumber').innerText = "This number is valid. You may continue.";
      }
    } catch(error) {
      console.error(`Red alert! We have an error: ${error.message}`);
    }
  });
});
```

Now in our `checkNumber()` function, if the number is `NaN` or below 0, we return an `Error`.

An `Error` is a built-in JavaScript object. There are a number of different types of errors that we could specify; for instance, instead of creating a new `Error` object, we could create a new `RangeError`. In fact, a `RangeError` would make more sense here because it's more specific. The documentation for JavaScript's `Error` object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) states that a `RangeError` represents "an error that occurs when a numeric variable or parameter is outside of its valid range", and that's exactly the case here. Later on in our `try...catch` we incorporate a `RangeError`.

Also notice that we pass a string value into the `Error` object:

```
if (isNaN(number) || number < 0) {
  return new Error("Not a valid number!");
}
```

We should always pass a value into any `Error` objects we create. When the error is raised, we can see this value by looking at its `message` property. Because errors can be very difficult to pinpoint in a larger application, the added detail is essential for debugging.

The next update we make is to our `try...catch` block:

```
try {
  const isNumberValid = checkNumber(inputtedNumber);
  if (isNumberValid instanceof Error) {
    console.error(isNumberValid.message);
    throw RangeError("Not a valid number!");
  } else {
    console.log("Try was successful, so no need to catc
h!");
    document.querySelector('#displayNumber').innerText = "T
his number is valid. You may continue.";
  }
} catch(error) {
  console.error(`Red alert! We have an error: ${error.messa
ge}`);
}
```

Now we save the result from calling the `checkNumber()` function in the `isNumberValid` variable, and we run that variable through a conditional. In our `if` statement, we check to see if `isNumberValid` is an `instanceof Error`. If so, our application will throw a `RangeError`.

We have a new JavaScript **operator** here: `instanceof`. The `instanceof` operator is specifically used to check the type of a JavaScript object. It does this by looking at the prototype chain of the object, which is a topic beyond the scope of this lesson.

We can test out `instanceof` in the DevTools console:

```
> let error = new Error();
> let error2 = new RangeError();
> error instanceof Error;
true
> error2 instanceof Error;
true
```

Both of the last two expressions return `true`. Note that while `error2` is a `RangeError`, this object type is also an `Error` as well.

Also notice that we use `console.error()` in our first `if` statement:

```
try {
  ...
  if (isNumberValid instanceof Error) {
    console.error(isNumberValid.message);
    throw RangeError("Not a valid number!");
  }
} catch {
  ...
}
```

You may have stumbled across `console.error` before, but if you haven't, it operates in a similar fashion to `console.log`. The only difference is that the message is outlined in red. There's also `console.warn`, which is generally used for notifications about deprecated functionality.

Inside our `console.error` message, we log the `error.message`. If we hadn't passed a string into our `Error` object before, the `message` property would be `undefined` and we'd be depriving ourselves of useful information for debugging.

Next, we `throw` a `RangeError` to ensure that control moves to the `catch` block. Keep in mind that `throw` allows developers to define exceptions in an application.

For instance, JavaScript itself doesn't care if we call `payBalance()` twice, charging our customers double in the process. To actually catch that behavior, we'd need to write and throw a custom exception.

Also keep in mind that if we `throw` an error outside of a `try...catch` block, the program will terminate. That's not really what we want, however. Instead, our application should be able to handle the error gracefully without terminating (unless it's absolutely necessary to terminate).

Our `catch` block is the same as it was before:

```
try {
  ...
} catch(error) {
  console.error(`Red alert! We have an error: ${error.messa
ge}`);
}
```

Take note that our `catch` block could handle exceptions in a number of ways. The most obvious (and passive) is to log the error. However, since control has moved to the `catch` block, we could technically run any code we want, including code that allows us to handle the error gracefully without terminating, like displaying a message to the user about the error.

Remember: Only Use `try...catch` for Exceptional Errors

It's very simple to incorporate `try...catch` blocks. In fact, it's easy enough that it can be very tempting to start using these blocks to handle many errors, including unexceptional ones. However, this is a mistake for a number of reasons. When developers see a `try...catch` block, they will assume it's for handling serious exceptions. Using `try...catch` blocks in other cases can be confusing. For instance, why would we use `try...catch` for user input when validations are used for that exact purpose?

`try...catch` can also result in a performance hit. While this usually won't be an issue, it's important to consider, particularly when an application has a long and resource-intensive stack trace that logs all of the function calls that led up to the error.

Just as importantly, our code would become both unreadable and very painful to write if we wrapped everything in a `try...catch` block. Think about going through the scanner at the airport. You only need to do that once, when you're going into the terminal, not every time you go to the bathroom!

While the basics of exception handling are relatively easy, knowing when to use exception handling is a more advanced concept that comes with practice. As you build out classwork projects, consider situations where your application might have serious exceptions. You may find opportunities to practice using them even before we incorporate them in a future lesson.

Previous (/intermediate-javascript/asynchrony-and-apis/the-devtools-network-tab)
Next (/intermediate-javascript/asynchrony-and-apis/team-week-project-pitch)
Lesson 11 of 33
Last updated more than 3 months ago.

disable dark mode