Understanding Errors

Foundations Course

Introduction

Reading and understanding error messages is a requirement as a developer. At first glance, many beginners shrink away from error messages as they appear to be "scary" and difficult to understand because they include terms one may not be familiar with.

However, error messages provide developers with a treasure trove of knowledge, and tell you everything you need to know about how to resolve them! Being able to parse error messages and warnings without fear will enable you to effectively debug your applications, receive meaningful help from others, and empower yourself to push forward when faced with an error.

Lesson overview

This section contains a general overview of topics that you will learn in this lesson.

- Name at least three kinds of JavaScript errors.
- Identify two parts of an error message that help you find where the error originates.
- Be able to understand how to research and resolve errors.

The anatomy of an error

An error is a type of object built into the JS language, consisting of a name/type and a message. Errors contain crucial information that can assist you in locating the code responsible for the error, determining why you have this error, and resolving the error.

Let's assume we have written the following code in a `script.js` file linked to an HTML page:

```
const a = "Hello";
const b = "World";

console.log(c);
```

This code will run, but it will generate an error. In technical terms, this is called "throwing" an error. The first part of an error displays the type of error. This provides the first clue as to what you're dealing with. We'll learn more about the different error types later in the lesson. In this example, we have a `ReferenceError`.

```
▶ Uncaught ReferenceError: c is not defined                    script.js:3
    at script.js:3
```

A `ReferenceError` is thrown when one refers to a variable that is not declared and/or initialized within the current scope. In our case, the error message explains that the error has occurred because `c is not defined`.

Different errors of this type have different messages based on what is causing the `ReferenceError`. For example, another message you may run into is `ReferenceError: can't access lexical declaration 'X' before initialization`.

As we can see, this points to a completely different reason than our original `ReferenceError` above. Understanding both the error type and the error message is crucial to comprehending why you are receiving the error.

The next part of an error gives us the name of the file in which you can find the error (in this case, our `script.js`), and also the line number.

This allows you to easily navigate to the problematic line in your code. Here, the error originates from the fourth line of `script.js`, which is displayed as a link under the error message with the text `at script.js:4`. If you click this link, most browsers will navigate to the exact line of code and the rest of your script in the Sources tab of the Developer Tools.

Sometimes your browser's console will also display the column (or character) in the line at which the error is occurring. In our example, this would be `at script.js:4:13`.

Another important part of an error is the **stack trace**. This helps you understand when the error was thrown in your application, and what functions were called that led up to the error. So, for example, if we have the following code:

```
const a = 5;
const b = 10;

function add() {
  return c;
}

function print() {
  add();
}
print();
```

Our function `print()` should call on `add()`, which returns a variable named `c`, which currently has not been declared. The corresponding error is as follows:

```
● ▸ Uncaught ReferenceError: c is not defined                  script.js:5
      at add (script.js:5)
      at print (script.js:9)
      at script.js:12
```

The stack trace tells us that:

1. `c is not defined` in scope of `add()`, which is declared on line 5.
2. `add()` was called by `print()`, which was declared on line 9.
3. `print()` itself was called on line 12.

Thus the stack trace lets you trace the evolution of an error back to its origin, which here is the declaration of `add()`.

Common types of errors

These are some of the most common errors you will encounter, so it's important to understand them.

Syntax error

A syntax error occurs when the code you are trying to run is not written correctly, i.e., in accordance with the grammatical rules of JavaScript. For example this:

```
function helloWorld() {
  console.log "Hello World!"
}
```

will throw the following error, because we forgot the parentheses for `console.log()`!

```
● Uncaught SyntaxError: Invalid or unexpected token
```

Reference error

We covered reference errors in the first example in this lesson, but it's important to remember that these arise because whatever variable you are trying to reference does not exist (within the current scope) - or it has been spelled incorrectly!

Type error

These errors are thrown for a few different reasons:

Per MDN, a `TypeError` may be thrown when:

> - ***an operand or argument passed to a function is incompatible with the type expected by that operator or function;***
> - ***or when attempting to modify a value that cannot be changed;***
> - ***or when attempting to use a value in an inappropriate way.***

Say we have two strings that you would like to combine to create one message, such as below:

```
const str1 = "Hello";
const str2 = "World!";
const message = str1.push(str2);
```

```
● ▸ Uncaught TypeError: str1.push is not a function            script.js:3
      at script.js:3
```

Here, we get a `TypeError` with a message stating that `str1.push is not a function`. This is a common error message that confuses learners because you might know that `.push()` is certainly a function (for example, if you have used it to add items to arrays) before.

But that's the key - `.push()` is not a String method, it's an Array method. Hence, it is "not a function" that you can find as a String method. If we change `.push()` to `.concat()`, a proper String method, our code runs as intended!

A good note to keep in mind when faced with a `TypeError` is to consider the data type you are trying to run a method or operation against. You'll likely find that it is not what you think, or the operation or method is not compatible with that type.

Tips for resolving errors

At this point, you might be wondering how we can resolve these errors.

1. We can start by understanding that the error message is your friend - not your enemy. Error messages tell you *exactly* what is wrong with your code, and which lines to examine to find the source of the error. Without error messages it would be a nightmare to debug our code - because it would still not work, we just wouldn't know why!
2. Now, its time to Google the error! Chances are, you can find a fix or explanation on StackOverflow or in the documentation. If nothing else, you will receive more clarity as to why you are receiving this error.
3. Use the debugger! As previously mentioned, the debugger is great for more involved troubleshooting, and is a critical tool for a developer. You can set breakpoints, view the value of any given variable at any point in your application's execution, step through code line by line, and more! It is an extremely valuable tool and every programmer should know how to use it.
4. Make use of the console! `console.log()` is a popular choice for quick debugging. For more involved troubleshooting, using the debugger might be more appropriate, but using `console.log()` is great for getting immediate feedback without needing to step through your functions. There are also other useful methods such as `console.table()`, `console.trace()`, and more!

Errors vs. warnings

Lastly, many people are met with warnings and treat them as errors. Errors will stop the execution of your program or whatever process you may be attempting to run and prevent further action. Warnings, on the other hand, are messages that provide you insight on potential problems that may not necessarily crash your program at runtime, or at all!

While you should address these warnings if possible and as soon as possible, warnings are not as significant as errors and are more likely to be informational. Warnings are typically shown in yellow, while errors are typically shown in red. Though these colors are not a rule, frequently there will be a visual differentiation between the two, regardless of the platform you are encountering them on.

Assignment

1. Now, it's time to go through the documentation! Learn more about the ReferenceError, the SyntaxError and the TypeError from the MDN Docs. Don't worry about fully understanding all the documentation right now; the goal is to familiarize yourself with the concepts. The examples use `try...catch` statements, which execute the code within the "try" block. If an error occurs, it is automatically caught by the "catch" block. This allows you to tackle errors before they terminate the script, allowing you to handle them appropriately within the "catch" block. For now, just remember that "try...catch" statements exist and that they will become useful as you progress through the curriculum.
2. Work through "What went wrong? Troubleshooting JavaScript". Be sure to download their starter code that has intentional errors.

Knowledge check

The following questions are an opportunity to reflect on key topics in this lesson. If you can't answer a question, click on it to review the material, but keep in mind you are not expected to memorize or master this knowledge.

- What are three reasons why you may see a TypeError?
- What is the key difference between an error and a warning?
- What is one method you can use to resolve an error?