Checkpoint 2

Error types

It may be hard to believe, but software developers often spend as much time fixing problems, or *bugs*, in their code as they do writing new code. And when there's a bug in your code, you'll likely encounter an error message. There are many error types and messages that you'll see throughout your career as a developer. In this checkpoint, you'll learn about some of the most common ones.

Outline

By the end of this checkpoint, you will be able to do the following:

- Recognize the differences between the three most common JavaScript error types
- Solve bugs by using error messages

Runtime errors

Time and time again, you'll receive errors in your console. It will be a regular occurrence—after all, bugs happen! When you see an error, it will probably read something like this:

```
SyntaxError: function statement requires a name
```

Take a moment to examine the different pieces of this message. The first part of the message, before the colon `:`, refers to the *error type*. The information that follows the colon `:` is a description of the error that has occurred. In the example above, the error description is `function statement requires a name`.

In the example above, the error type is `SyntaxError`. You will encounter three common error types in JavaScript, which you'll learn about below:

1. `ReferenceError`
2. `SyntaxError`
3. `TypeError`

Outline

Reference errors

Take a look at the following code. Can you spot the error?

```
const customerName = "Alfie Lee";
console.log(customrName);
```

As you probably noticed, the variable `customerName` was misspelled when it was inputted into the `console.log()` statement. Where the code should read `customerName`, it actually reads `customrName`. And though it may seem like a minor issue, using consistent spelling in your code is essential. When you run this code, it will reveal an error.

This is an example of a `ReferenceError`. The error that will surface because of this misspelling will look like the following:

```
ReferenceError: customrName is not defined
```

A *reference error* tells you that some variable being referenced does not exist. You can fix this kind of error by finding the offending reference and checking for the correct name. But a `ReferenceError` could also surface if the variable cannot be accessed at all. Can you identify this issue in the code below?

```
function addSalesTax(total, salesTax) {
  let result = total * (1 + salesTax);
  return result;
}

console.log(result);
```

Outline

In the above example, the `result` variable *does* exist, but it exists *only* inside the `addSalesTax()` function. Attempting to access that variable outside of that function would lead to a `ReferenceError`. (This is because of a concept called *scope*, which will be covered later in this module.)

Syntax errors

Another common error type is the `SyntaxError`. *Syntax errors* occur when some part of the predefined JavaScript syntax is used incorrectly, such as a character is used twice or not used at all. These errors are common for developers due to the simple fact that coding uses many

symbols and characters that people don't otherwise use. For instance, because curly brackets `{}` are unusual in regular typing, it can be easy to forget one.

Take a look at the following code. Can you find what's missing?

```
function printWelcome () {
  console.log("Welcome to our store!";
}
```

Above, you'll see that the `console.log()` statement is missing a closing parenthesis `)`. The error that would surface because of the above code would be as follows:

```
SyntaxError: missing ) after argument list
```

Outline

Although that message is fairly straightforward, it often isn't that simple. It can be challenging to actually resolve a `SyntaxError` with longer, more complex code. In the case of a `SyntaxError`, you often just need to look through your code patiently and carefully for the missing or extra symbol. Correctly indenting your code and using a code formatter can be useful for identifying (and preventing) issues like this. For example, take a look at the following code. Can you find the error? It will likely be pretty difficult!

```
function openInstructions (weather, temperatureInCels
if (weather && temperatureInCelsius) {
  if (weather === "sunny") {
  if (temperatureInCelsius > 20) {
    return "Set up the patio and put out umbrellas. (
```

```
    } else { return "Set up the patios, umbrellas optic
    } else if (weather === "rainy") {
      if (temperatureInCelsius > 10) {
        return "Open indoor windows slightly.";
      } else { return "Keep windows closed." } }
    }
  } else {
    return "Please set the `weather` and `temperatureIr
  }
  }

  openInstructions("sunny", 18);
```

If you were to run the above code, you would receive this message in the console:

```
SyntaxError: expected expression, got '}'
```

Outline

That's not very helpful, and you'd have to do the legwork of resolving the issue. This is one of the many reasons that it's important to write your code legibly.

Do this

Solve the syntax error

Copy the above code sample into your text editor, and correctly indent and space the code until you find the issue that's causing the `SyntaxError`. How might you avoid a problem like this in the future?

Type errors

And finally, you'll learn about the `TypeError`. *Type errors* occur when you misuse a data type in JavaScript, meaning that an operation can't be performed. One of the most common ways that this error will occur is through a situation like this:

```
price.trim(); //> TypeError: price.trim is not a func
```

As it turns out, the `trim()` function, when called on a string, removes extra whitespace from the beginning and the end of that string. So in this case, why would `price.trim()` not be a function? Well, this error is likely telling you that `price` isn't actually a string.

Take a look at the whole picture to better understand what's going on.

Outline

```
let price = 9.99;
price.trim(); //> TypeError: price.trim is not a func
```

You might expect this error to tell you that `price` is not the right data type, rather than telling you that `trim()` isn't a function. To better understand why the error is described the way it is, try running the following code:

```
let price = 9.99;
console.log(price.trim); //> undefined
```

You may be surprised to find that calling `trim` as a *property* on `price` does not fail—instead, it returns an `undefined`. But when you try to *invoke* the `undefined` property, that is when you get your error.

Silent errors

The three errors described above are caught and revealed to you when you run your JavaScript code. However, it's also possible for errors to occur as you are writing code but not immediately surface when you run it.

For example, take a look at the following code. Do you see any problems?

```
function formatPrice(priceInCents) {
  let formattedPrice = "$" + (priceInCents / 100).set
  return formattedPrice;
}
```

You may not notice any issues with this code right off the bat. And even when you run this code sample, no errors will surface. It is only when you *invoke* the function that you will see the error:

```
TypeError: (priceInCents / 100).setFixed is not a fur
```

In this case, the function uses `setFixed()`, which is *not* a function, instead of `toFixed()`, which *is* a function. When working with complex applications with multiple functions, you'll need to be aware of silent errors like this one. Always test out your functions after you make a change to make sure that everything is running as it should be.

Outline

Checkpoint

This checkpoint will be autograded. Please click the link below to open your assignment in a new tab. Once you complete the assignment, you will see a button allowing you to submit your answers and move on to the next checkpoint.

Your work

03.22.21 **Approved**

Completed

Next checkpoint

Outline

How would you rate this content?

Report a typo or other issue

Go to Overview