r/workspace/learn-javascript-debugging-code/app.js:1

"my\_name".capitalize();

^

TypeError: "my\_name".capitalize is not a function

...

Alternatively, you might have finally finished implementing a complex function only to find it doesn’t work at all as you expected:

console.log('The square root of 4 is ' + squareRoot(4));

=> 'The square root of 4 is 1.87878787'

If any of these things happen to you, don’t fret! In our article [Thinking About Errors Differently in Your Code](https://news.codecademy.com/errors-in-code-think-differently/), you discovered that these experiences are normal for all developers, no matter how experienced they are. A key skill developers possess is knowing how to debug their code when issues occur.

This lesson will build on that article, demonstrating how to implement those skills in real-life JavaScript situations. If you haven’t had a chance to read the article yet, we highly encourage you to do so right now.

Alright, are you ready to begin your transition from frustrated coder to bug squasher extraordinaire? Let’s get started!

**Error Stack Traces**

We’ll start this lesson by taking a closer look at the most straightforward way to know your code isn’t working as expected: errors!

You might recognize errors as the scary red text that appears on your screen when you try to run broken code. A piece of software, called a *compiler*, is trying to translate your code so that your computer can understand and run it. However, the compiler is coming across a piece of code that it can’t interpret. As a result, it throws an error back to you to let you know that it has to stop and why.

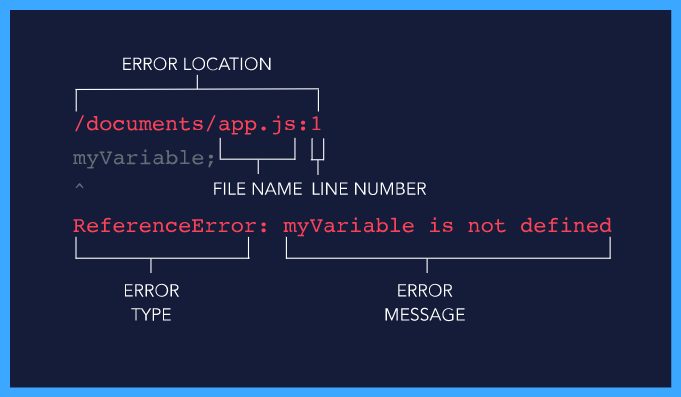
This information is logged as an **error stack trace** — a printed message containing information about where the error occurred, what type of error was thrown, and a description of the error.

Take a look at the diagram to the right to see where you can find all of this information.

As a new programmer, it’s easy to see an error and be overcome with a sense of dread and self-doubt. However, you’ll soon realize, errors are the best kind of bug you can hope for, since they tell you exactly what went wrong and where.

**Instructions**

Take a look at the diagram to the right. When you’re ready, move on to the next exercise to read a real error stack trace using this information.



# Reading Error Stack Traces

Now that we know what information we can get from an error stack trace, let’s take a look at an example.

/home/ccuser/workspace/learn-javascript-debugging-code/app.js:1

myVariable;

^

ReferenceError: myVariable is not defined

...

Using this stack trace, let’s answer three questions you should ask yourself every time you want to debug an error:

1. **In what line did the error occur?** You can almost always find this information on the first line of the stack trace in the following format <file path>/<file name>:<line number>. In this example, the location is app.js:1. This means that the error was thrown in the file named app.js on line 1.
2. **What type of error was thrown?** The first word on the fifth line of the stack trace is the type of error that was thrown. In this case, the type of error was ReferenceError. We will discuss this error type in the next exercise.
3. **What is the error message?** The rest of the fifth line after the error type provides an error message, describing what went wrong. In this case, the description is myVariable is not defined.

You might notice in this example we skipped explaining a few lines in the stack trace and only included the beginning of the stack trace. A large part of debugging errors is recognizing which pieces of information are useful and which ones aren’t. For now, you will succeed by focusing on just the first and fifth lines of your error stack traces.

**Instructions**

**1.**

Near the beginning of the text file in the code editor, you will find an example stack trace. On line 14 of the text file, fill in the name of the file throwing the error in this stack trace.

If you are unsure how to fill in the answer, feel free to check out the hint.

Hint

The file name can be found near the end of the first line of the error stack trace.

You can fill in the answer to the question like so:

From what file was this error thrown?

1 - Answer: file.js

**2.**

On line 16 of the text file, fill in the line number throwing the error in the example stack trace.

Hint

The line number can be found at the end of the first line of the error stack trace.

**3.**

On line 18 of the text file, fill in the type of the error being thrown in the example stack trace.

Hint

The error type can be found at the beginning of the fifth line of the error stack trace.

**4.**

On line 20 of the text file, fill in the description of the error being thrown in the example stack trace.

Hint

The error description can be found after the error type on the fifth line of the error stack trace.

Based on the following error stack trace, fill in the answers to the questions below:

------------------------------------------------------------------

/home/ccuser/workspace/learn-javascript-debugging-code/script.js:5

if (numberSum > total;) {

^

SyntaxError: Unexpected token ;

------------------------------------------------------------------

From what file was this error thrown?

1 - Answer: script.js

On what line of that file was this error thrown?

2 - Answer: 5

What type of error was thrown in this stack trace?

3 - Answer: SyntaxError

What is the description of the error in this stack trace?

4 - Answer: Unexpected token ;

**JavaScript Error Types**

Now that you can identify the type of error from an error stack trace, you might be wondering what the different types of errors mean.

Here are three common error types:

* **SyntaxError**: This error will be thrown when a typo creates invalid code — code that cannot be interpreted by the compiler. When this error is thrown, scan your code to make sure you properly opened and closed all brackets, braces, and parentheses and that you didn’t include any invalid semicolons.
* **ReferenceError**: This error will be thrown if you try to use a variable that does not exist. When this error is thrown, make sure all variables are properly declared.
* **TypeError**: This error will be thrown if you attempt to perform an operation on a value of the wrong type. For example, if we tried to use a string method on a number, it would throw a TypeError.

There are seven types of built-in JavaScript errors in total. You can find more information about all of them at the [MDN JavaScript Error documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error). Whenever you are confronted with an error you can’t comprehend, consulting this documentation is a great place to start.

**Instructions**

**1.**

Fill in the answer to the question on line 2.

Hint

Was myVariable defined anywhere in the code? What type of error gets thrown when an undefined variable is used?

**2.**

Fill in the answer to the question on line 6.

Hint

What type of data type is currently stored in myString? What type of error gets thrown when you try to use a data type as if it were a different data type?

If you want to check what the .substring() method does, check out the [MDN documentation on .substring()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substring).

**3.**

Fill in the answer to the question on line 9.

Hint

Can you find the typo on line 8? What type of error gets thrown when a line of code contains a typo?

myVariable++;

// 1 - What type of error will be thrown on the line above:

ReferenceError

const myString = 42;

myString.substring(0);

// 2 - What type of error will be thrown on the line above:

TypeError

const myRandomNumber; = Math.random();

// 3 - What type of error will be thrown on the line above:

SyntaxError

**Debugging Errors**

Here’s a process for efficiently working through your code’s errors one by one:

1. Run your code. Using the first error’s stack trace, identify the error’s type, description, and location.
2. Go to the file name and line number indicated by the error stack trace. Using the error type and description, identify the bug in your code.
3. Fix the bug and re-run your code.
4. Repeat steps 1-3 until your code no longer throws any errors.

While these steps may seem simple, it can be easy to get overwhelmed by errors in practice. Using these steps, you can tackle your errors one at a time and soon will have your code running error-free.

**Instructions**

**1.**

The code in the code editor currently contains three errors. Using the above steps, locate and fix all three errors in the code. Don’t be afraid of any error messages that appear, you are triggering them on purpose! Just take your time locating them and fixing them one at a time.

If you get stuck, feel free to consult the hint.

**Note**: The error messages in the terminal to the right may wrap to a new line based on your screen size. As a result, what you think is the end of a line might only be partway through it. Read through all of the error stack trace to ensure you find the information you’re looking for.

Hint

The first error is a SyntaxError saying that there is an 'Unexpected token {' on line 1 of app.js. Can you find a typo in the function declaration on line 1?

The second error is a SyntaxError saying there is an 'Unexpected token ;' on line 4 of app.js. Can you find the semicolon typo in the if statement condition on line 4?

The third error is a ReferenceError saying 'numberSum is not defined' on line 4 of app.js. Can you find the typo where this variable was supposed to be defined? How can we fix this code so the variable is defined properly?

function isSumBigger(number1, number2, total ){

  const sum = number1 + number2;

  if (sum > total) {

    return true;

  } else {

    return false;

  }

}

// Should return true

console.log(isSumBigger(1, 3, 2)+' returns: ' + isSumBigger(1, 3, 2));

// Should return false

console.log(isSumBigger(1, 3, 5)+' returns: ' + isSumBigger(1, 3, 5));

# Locating Silent Bugs

Errors thrown by the computer are really useful because they identify the bug type and location for you right away. However, even if your code runs error-free, it is not necessarily bug-free.

You may find that your code is consistently returning incorrect values without throwing any errors. A lack of thrown errors does not mean your code logic is completely correct.

An incredibly powerful tool for locating bugs is a method you likely learned very early on in your JavaScript journey: console.log()!

By adding print statements to our code, we can identify where things have gone wrong.

Let’s try using console.log() to do some debugging.

**Instructions**

**1.**

In the code editor, we’ve given you a function that is supposed to take a string containing a single word as an argument and return the capitalized version of that word. If the string contains multiple words (contains a space character, ' '), it should return null.

Currently, this code is not working as expected. Take a look at our tests on lines 16-19. We expect the first function call to return 'Hey' and the second function call to return null. Run the code to see that it is not working as expected but also not throwing any errors.

**2.**

Let’s use console.log() to debug this code! Start by adding a console.log() statement to line 2 and printing out the word being passed into the function. This will help us ensure our function is being called properly and the word variable is set as we expect at the start of the function.

**3.**

Looking at the console output, both 'hey' and 'hey ho' are being printed as we expected. No bugs yet.

Remove the console.log() statement from the beginning of the function. Then add the following console.log() statement inside the if statement to see if it is working properly:

console.log('Word value inside of if statement: ' + word);

The behavior we want is for 'hey ho' to enter the if block and return null and for 'hey' to skip the if block and get capitalized.

**4.**

Look at the console output. 'hey' is being printed from inside the if statement block and 'hey ho' is not. This is the opposite behavior we expected. We found a bug!

Looking at our if statement, we accidentally included a ! that is negating our condition. It is saying that if a word does not have multiple words, go into the if block, and if it does, move past it.

Delete the ! character from the beginning of the if statement condition to fix the bug, then re-run our code.

Our function now appears to be working as expected, so there is no need to continue stepping through the rest of the function.

function capitalizeASingleWord(word) {

  if (word.match(' ')) {

      console.log(word);

    return null;

  }

  let firstLetter = word.charAt(0);

  const restOfWord = word.slice(1);

  firstLetter = firstLetter.toUpperCase();

  return firstLetter + restOfWord;

}

// Should return "Hey"

console.log("capitalizeASingleWord('hey') returns: " + capitalizeASingleWord('hey'));

// Should return null

console.log("capitalizeASingleWord('hey ho') returns: " + capitalizeASingleWord('hey ho'));

**Debugging with console.log()**

Let’s synthesize our workflow from the previous exercise into a reusable set of debugging steps.

1. Go to the beginning of the malfunctioning code. Print out all starting variables, existing values, and arguments using console.log(). If the values are what you expect, move on to the next piece of logic in the code. If not, you have identified a bug and should skip to step 3.
2. After the next piece of logic in your code, add console.log() statements to ensure updated variables have the values that you now expect and that the block of code is being executed. If that logic is executing properly, continue repeating this step until you find a line not working as expected, then move to step 3.
3. Fix the identified bug and run your code again. If it now works as expected, you’ve finished debugging! If not, continue stepping through your code using step 2 until it does.

This might seem like a lot of printing, but once you get into the routine of it, it will be far faster than trying to stare at your code until you find your bugs. Let’s try this debugging process again in practice.

**Instructions**

**1.**

In the code editor, we have included a broken function for you to debug. This function is supposed to take two strings, compare the first letter of each, and return the string which comes later alphabetically. If the first letters of both strings are the same, it should return null.

At the bottom of the file, we’ve included three test cases and their expected outputs. If you run the code, you will see that they are not working as expected.

Using the console.log() debugging mindset laid out above, walk through the code, locate the three bugs we have included for you, and fix them.

This is a big task, so feel free to consult the hint for suggestions on where to place your console.log() statements and how to fix the code.

Hint

**Bugs 1 and 2**: Print out the starting values of firstLetter1 and firstLetter2 and run the code. Are these actually the first letters of the strings? How can we modify the calls to .charAt() to make them the first letters?

**Bug 3**: Add print statements to the insides of the if/else blocks and run the code. Are the proper values being returned based on the block being executed? How can we change the conditions to return the proper values?

// Returns the string whose first letter is later in the alphabet. If both first letters are equal, returns null.

function getLaterFirstLetter(string1, string2) {

  const firstLetter1 = string1.charAt(0);

  const firstLetter2 = string2.charAt(0);

  if (firstLetter1 === firstLetter2) {

    return null;

  } else if (firstLetter1 < firstLetter2) {

    return string2;

  } else {

    return string1;

  }

}

// Should return "blueberry"

console.log("getLaterFirstLetter('avocado', 'blueberry') returns: " + getLaterFirstLetter('avocado', 'blueberry'));

// Should return "zebra"

console.log("getLaterFirstLetter('zebra', 'aardvark') returns : " + getLaterFirstLetter('zebra', 'aardvark'));

// Should return null

console.log("getLaterFirstLetter('astro', 'afar') returns: " + getLaterFirstLetter('astro', 'afar'));

# Finding Documentation

Sometimes once you’ve tracked down a bug, you might still be confused on how to fix it! Whenever you want to know more about how JavaScript works and what it can do, the best place to go is **documentation**. You can find the JavaScript documentation at the [MDN JavaScript web docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript).

The MDN JavaScript web docs are a powerful resource, but they can be overwhelming because they cover so much information. We encourage you to explore the docs, but often the fastest way to access a specific part of the docs you’re interested in is to Google it.

For example, if we wanted more information on the Number object’s .isNan() method, we could Google “MDN isNan” and then click the link to the MDN page. If we were looking to see a list of all of the String built-in methods, we might Google “MDN String”, click the link to MDN, and then scroll down to the “Methods” section of the documentation.

There are many ways to get to the documentation you are looking for. Find the one that works best for your workflow.

**Instructions**

**1.**

In the code editor, we’ve included a function that is supposed to take a string and return a version of that string repeated twice. For example, you can see in the test case at the bottom of the file, we expect doubleString('echo') to return 'echoecho'.

However currently, when we run our file, nothing happens. Clearly we are using the .repeat() method incorrectly. Let’s find the MDN documentation for String.repeat().

Find the link to the documentation for String.repeat() and paste it to the end of the comment on line 1.

Hint

Try googling “JavaScript string repeat documentation” and finding the MDN link from the search results.

**2.**

Now that we’ve found the documentation for String.repeat(), let’s read the documentation to see what we’re doing wrong. Take a look at the [parameters section of the documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/repeat#Parameters). Can you see why our string isn’t getting repeated?

Using this information, fix the .repeat() call in our code.

Hint

The documentation notes that there is a required parameter count that must be provided to the .repeat() method. Add the correct value of this argument to the code in the code editor.

// Link to String.repeat() documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String/repeat

function doubleString(string) {

  return string.repeat(2);

}

// Should return 'echoecho'

console.log("doubleString('echo') returns: " + doubleString('echo'));

// Returns whether or not the provided string contains a substring of "cake" in it.

function containsCake(string) {

  return string.includes('cake');

}

// Should return true

console.log("containsCake('I think cake is my soul mate.') returns: " + containsCake('I think cake is my soul mate.'));

// Should return false

console.log("containsCake('Pie is certainly the coolest dessert.') returns: " + containsCake('Pie is certainly the coolest dessert.'));

**Debugging Review**

You just learned a lot of techniques for helping you get unstuck in all debugging situations. Congratulations! Let’s synthesize everything you learned into one debugging process.

1. **Is your code throwing errors?** If so, read the error stack trace for the type, description, and location of the error. Go to the error’s location and try to fix.
2. **Is your code broken but not throwing errors?** Walk through your code using console.log() statements. When unexpected results occur, isolate the bug and try to fix it.
3. **Did you locate the bug using steps 1 and 2, but can’t fix the bug?** Consult documentation to make sure you are using all JavaScript functionality properly. If you are still stuck, Google your issue and consult Stack Overflow for help. Read solutions or post your own Stack Overflow question if none exist on the topic.

It may take some time and practice, but this is how all developers work through their issues, so don’t give up, and you’ll be an expert in no time.

Congratulations again on learning all of these new techniques. Let’s give them one more round of practice so you can see how much progress you’ve made!

**Instructions**

**1.**

In the code editor, we have provided you a broken implementation of a function called isStringPerfectLength(). This function takes a string (string), a minimum string length (minLength), and a maximum string length (maxLength). It should return whether the provided string is within the range of the two lengths. More specifically, if it is longer than the minLength and shorter than the maxLength.

We have provided three example test cases at the bottom of the file. The first checks if the string 'Dog' is in the length range of 2 and 4. In this case, the function should return true because the length of the string is 3 which is between those two values. The next two examples fail. The first fails because the string is too long. The second fails because the string is too short.

Using the debugging process you learned in this lesson, outlined above, find and fix all the bugs in this code.

The code will throw errors at your right off the bat, but remember, this is a good thing! We believe you have all the skills now to debug this code. Don’t give up and show us what you’ve learned! We know you can do it.

function isStringPerfectLength(string, minLength, maxLength) {

  const stringLength = string.length;

  if (stringLength < minLength) {

    return false;

  } else if (stringLength > maxLength) {

    return false;

  } else {

    return true;

  }

}

// Should return true

console.log("isStringPerfectLength('Dog', 2, 4) returns: " + isStringPerfectLength('Dog', 2, 4));

// Should return false

console.log("isStringPerfectLength('Mouse', 2, 4) returns: " + isStringPerfectLength('Mouse', 2, 4));

// Should return false

console.log("isStringPerfectLength('Cat', 4, 9) returns: " + isStringPerfectLength('Cat', 4, 9));

# Introduction to Error Handling

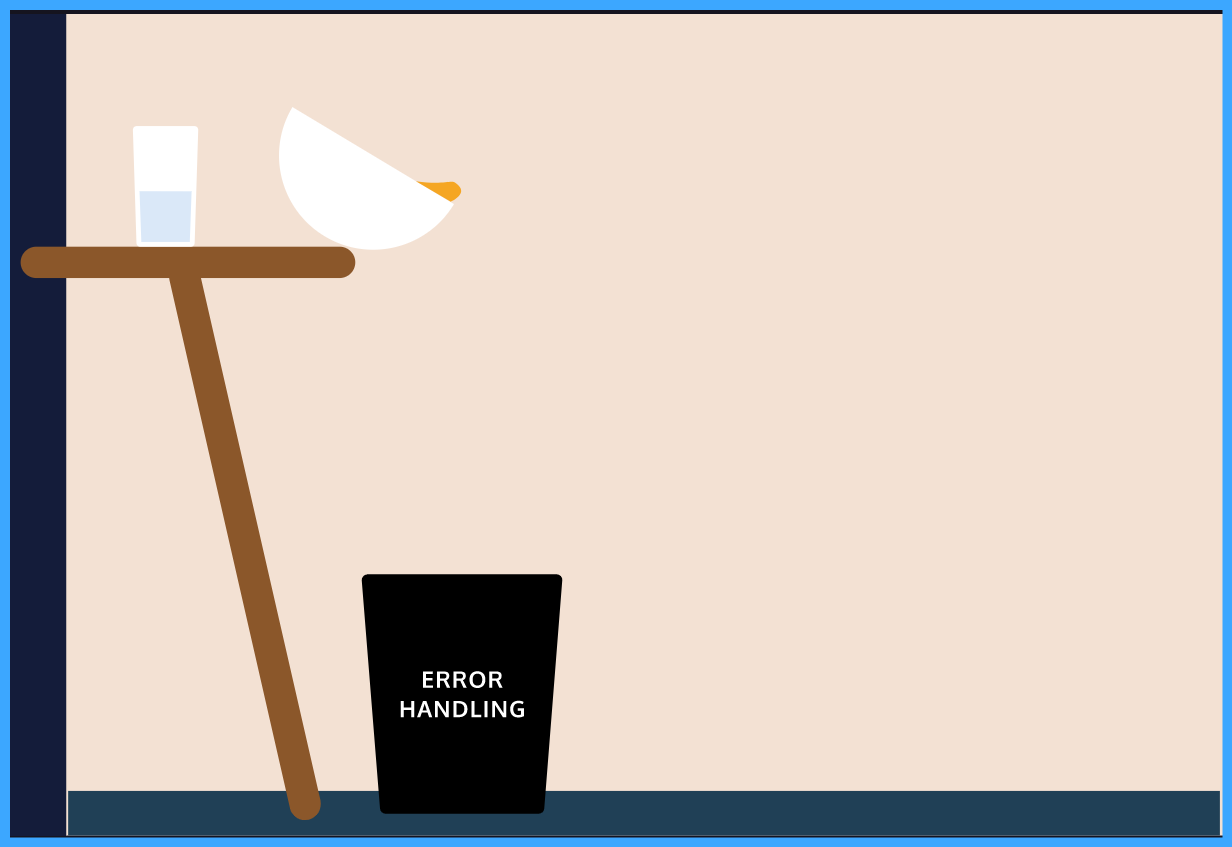
There are two categories of programming mistakes: those that don’t prevent our code from running and those that do.

Sometimes, we’ve written code that successfully returns a value but a different value from what we expected. Our program continues running, and we might not even realize anything went wrong until much later. It’s like making soup and accidentally adding sugar instead of salt. In the end we still have soup, but it might not be soup that we want to eat. We will not be focusing on these mistakes.

Rather, we’re going to focus on the errors that pop up when we’ve written code that causes our program to stop running, e.g. trying to reassign a const variable. Instead of returning anything, our program will not execute any more code past where the error occurred. For example, what if we tried to move our soup to the table but dropped it because it was too hot? Then our soup-making process is over— there would be no soup.

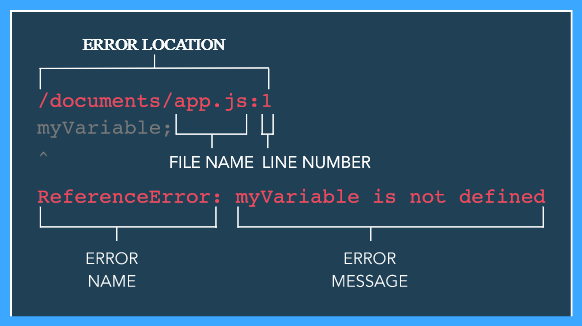
We can’t always stop errors before they occur, but we can include a backup plan in our program to anticipate and respond to the errors to ensure that our program continues running. Error handling is the process of programmatically anticipating and addressing errors. In JavaScript, we handle errors using the keywords try and catch. We try to move the soup to the table, making sure there’s someone or something nearby to catch the soup in case we drop it.

In this lesson we’ll learn more about errors and how to create a backup plan to allow our program to continue running. When you’re ready, let’s try to get a handle on these JavaScript errors!



**Runtime Errors**

Errors contain useful messages that tell us why our program isn’t working or why the error was *thrown*. When an error is thrown, our program stops running and the console displays the error message in red text like so:



When we execute code and a line of code throws an error, that error is referred to as a *runtime error*. In JavaScript, there are built-in error objects that have a name and message property which tell us what went wrong. Examples of built-in runtime errors include:

* ReferenceError: when a variable or function cannot be found.
* TypeError: when a value is not a valid type, see the example below:

const reminder = 'Reduce, Reuse, Recycle';

reminder = 'Save the world';

// TypeError: Assignment to constant variable.

console.log('This will never be printed!');

In the example above, when we try to reassign a constant variable reminder, the TypeError is thrown. Code that is written after a thrown runtime error will not be evaluated, so the console.log() statement will not be evaluated.

Let’s see for ourselves what happens when a runtime error is thrown.

**Instructions**

**1.**

In **main.js**, we have a function throwError() that will throw a ReferenceError. There are also two console.log() statements, one at the top of the file and one at the bottom.

Call throwError() below the comment that indicates to call it. Observe what code runs and what code doesn’t.

Hint

Make sure you call throwError() before the second console.log() statement. If done correctly, the first console.log() statement will print while the second console.log() statement won’t.

console.log('This message will be printed to the console.');

function throwError () {

  return notDefinedVar;

}

// Call throwError() below:

 throwError();

console.log('Because of the error, this will not printed!');

# Constructing an Error

JavaScript errors are objects that have a name and message property. Previously, we’ve seen how built-in errors alert us about common mistakes in our code. But, what if we need an error message that isn’t covered by the built-in errors? Let’s say we need to inform a user that a string passed in as an argument is too short with a custom message. Such a message isn’t covered by a built-in error, but we could use the Error function to make our own error object with a message that is unique to our program!

console.log(Error('Your password is too weak.'));

// Prints: Error: Your password is too weak.

The Error function takes an argument of a string which becomes the value of the error’s message property. In the code snippet above, we used the Error function to create an error object that has the message 'Your password is too weak.'.

You might also see errors created with the new keyword. Both methods will lead to the same functionality. Take a look:

console.log(new Error('Your password is too weak.'));

// Prints: Error: Your password is too weak.

Keep in mind that creating an error is not the same as throwing an error. A thrown error will cause the program to stop running. We cover how to throw our created errors in the next exercise!

**Instructions**

**1.**

At the top of **main.js** add a console.log() that prints an error with the message 'User missing name'.

After running the code, take note of the console.log() at the bottom of **main.js** and if that line of code still runs.

Hint

The argument passed to the Error() function becomes the message property of the new error. You may use the Error() function create the error object or use the new keyword with the Error().

console.log(Error('Message goes here'));

// OR

console.log(new Error('Message goes here'));

// Write your code below:

console.log(Error('User missing name'));

console.log('Will logging the error stop our program from running?');

# The throw Keyword

Creating an error doesn’t cause our program to stop — remember, an error must be thrown for it to halt the program.

To throw an error in JavaScript, we use the throw keyword like so:

throw Error('Something wrong happened');

// Error: Something wrong happened

When we use the throw keyword, the error is thrown and code after throw statement will not execute. Take for example:

throw Error('Something wrong happened');

// Error: Something wrong happened

console.log('This will never run');

After throw Error('Something wrong happened'); is executed and the error object is thrown, the console.log() statement will not run (just like when a built-in JavaScript error was thrown!).

In the next lesson we will cover how to handle an error so that the rest of our code can run!

**Instructions**

**1.**

Use the throw keyword to throw an error with the message 'Username or password do not match'

After you clear this checkpoint, try adding some code after your throw statement. Also try to throw a new error and other data types to see what happens!

Hint

To use the throw keyword, use the following syntax:

throw Error('Helpful message to pinpoint what went wrong!');

Or

throw new Error('Helpful message to pinpoint what went wrong!');

throw Error('Username or password do not match');

console.log('abc');

**The try...catch Statement**

Up to this point, thrown errors have caused our program to stop running. But, we have the ability anticipate and *handle* these errors by writing code to address the error and allow our program to continue running.

In JavaScript, we use try...catch statement to anticipate and handle errors. Take a look at example below:

try {

throw Error('This error will get caught');

} catch (e) {

console.log(e);

}

// Prints: This error will get caught

console.log('The thrown error that was caught in the try...catch statement!');

// Prints: 'The thrown error that was caught in the try...catch statement!'

Now, let’s break down what happened in the try...catch statement above:

* We have code that follows try inside curly braces {} known as the *try block*.
* Inside the try block we insert code that we anticipate might throw an error.
* Since we want to see what happens if an error is thrown in the try block, we throw an error with the message 'This error will get caught'.
* Following the try block is the catch statement which accepts the thrown error from the try block . The e represents the thrown error.
* The curly braces that follow catch(e) is known as the *catch block* and contains code that executes to handle the error.
* Since the error is caught, our console.log() after the try...catch statement prints 'The thrown error that was caught in the try...catch statement!'.

Generally speaking, in a try...catch statement, we evaluate code in the try block and if the code throws an error, the code inside the catch block will handle the error for us. The provided example just showcases how a try...catch statement works because we know an error is being thrown. Let’s first practice writing our own try...catch statement and afterwards we will go over a more practical usage of try...catch.

**Instructions**

**1.**

Create a try...catch statement that doesn’t have any code in the try block or the catch block. The catch statement will have accept e for the error object.

Hint

The syntax for a try...catch statement:

try {

// try block code

} catch (e) {

// catch block code

}

**2.**

In the try block, throw an error using the Error() function and pass into Error() a string containing a message of your choice!

Hint

To throw an error, use the throw keyword and the Error function:

throw Error('error message goes here');

Make sure you’re adding the code within the try block:

try {

throw Error('error message goes here');

} catch (e) {

// catch block code

}

**3.**

Inside the catch block, log the error to the console.

Hint

To log the error to the console, inside your catch block add a console.log() statement.

try {

throw Error('error message goes here');

} catch (e) {

console.log(e);

}

try {

throw Error('error message goes here');

} catch (e) {

  console.log(e);

}

**Handling with try...catch**

In the previous exercise we caught an error that we threw, but we can also use a try...catch statement to handle built-in errors that are thrown by the *JavaScript engine* that is reading and evaluating our code.

const someVar = 'Cannot be reassigned';

try {

someVar = 'Still going to try';

} catch(e) {

console.log(e);

}

// Prints: TypeError: Assignment to constant variable.

In the example above, we didn’t use the throw keyword to throw a custom error, rather we tried to re-assign a const variable and a TypeError was thrown. Then, in our catch block, we logged the error to the console.

Using a try...catch statement for built-in JavaScript errors is really beneficial when we need to use data from an external source that’s not written directly in our program.

Let’s say we expect to grab an array from a database but the information we get back is a string. In our program, we could have a function that only works on arrays. If that function was called with a string instead of an array we would get an error and our program would stop running!

However, we can use a try...catch statement to handle the thrown error for us which allows our program to continue running and we receive a message knowing what went wrong! Let’s see how we can implement this in our code.

**Instructions**

**1.**

In **main.js** there is a function capAllElements() that takes an array of elements and capitalizes each element.

Currently, it’s written in a way the function will execute regardless of what argument is passed in but if the argument isn’t an array, an error is thrown and our program stops running. Run your code to see what error shows up in the console.

Let’s handle that error:

* Put the existing arr.forEach(...) code in a try block.
* Add a catch statement after the try block and inside the catch block log the error to the console.

Hint

Inside the function body of capAllElements(), wrap the arr.forEach(...) in a try...catch statement:

function capAllElements(arr){

try {

arr.forEach((el, index, array) => {

array[index] = el.toUpperCase();

});

} catch (e) {

}

}

Then add the console.log() statement inside the catch block:

function capAllElements(arr){

try {

arr.forEach((el, index, array) => {

array[index] = el.toUpperCase();

});

} catch (e) {

console.log(e);

}

}

**Handling with try...catch**

In the previous exercise we caught an error that we threw, but we can also use a try...catch statement to handle built-in errors that are thrown by the *JavaScript engine* that is reading and evaluating our code.

const someVar = 'Cannot be reassigned';

try {

someVar = 'Still going to try';

} catch(e) {

console.log(e);

}

// Prints: TypeError: Assignment to constant variable.

In the example above, we didn’t use the throw keyword to throw a custom error, rather we tried to re-assign a const variable and a TypeError was thrown. Then, in our catch block, we logged the error to the console.

Using a try...catch statement for built-in JavaScript errors is really beneficial when we need to use data from an external source that’s not written directly in our program.

Let’s say we expect to grab an array from a database but the information we get back is a string. In our program, we could have a function that only works on arrays. If that function was called with a string instead of an array we would get an error and our program would stop running!

However, we can use a try...catch statement to handle the thrown error for us which allows our program to continue running and we receive a message knowing what went wrong! Let’s see how we can implement this in our code.

**Instructions**

**1.**

In **main.js** there is a function capAllElements() that takes an array of elements and capitalizes each element.

Currently, it’s written in a way the function will execute regardless of what argument is passed in but if the argument isn’t an array, an error is thrown and our program stops running. Run your code to see what error shows up in the console.

Let’s handle that error:

* Put the existing arr.forEach(...) code in a try block.
* Add a catch statement after the try block and inside the catch block log the error to the console.

Hint

Inside the function body of capAllElements(), wrap the arr.forEach(...) in a try...catch statement:

function capAllElements(arr){

try {

arr.forEach((el, index, array) => {

array[index] = el.toUpperCase();

});

} catch (e) {

}

}

Then add the console.log() statement inside the catch block:

function capAllElements(arr){

try {

arr.forEach((el, index, array) => {

array[index] = el.toUpperCase();

});

} catch (e) {

console.log(e);

}

}

function capAllElements(arr){

  try {

    arr.forEach((el, index, array) => {

      array[index] = el.toUpperCase();

    });

  } catch (e) {

    console.log(e);

  }

}

capAllElements('Incorrect argument');

**Error Handling Review**

Great job with handling errors!

In this lesson we went over:

* How mistakes in programming leads to errors.
* Why errors are useful for developers.
* Errors will prevent a program from executing unless it is handled.
* How to create an error using the Error() function.
* How to throw an error object using the throw keyword.
* How to use the try...catch statement to handle thrown errors.
* Evaluating code in a try block to anticipate errors.
* Catching the error in a catch block to allow our program to continue running.
* Why the try...catch statement would be useful in a program.

Now you have the ability to create code that doesn’t break when an error is thrown!

**Instructions**

If you want to challenge yourself:

* Force different built-in error object to be thrown in a try...catch statement and see when the messages are different.
* Test out what different data types you can throw.
* Use try...catch for a function that accepts user input to allow errors to be thrown but your program to continue running.