Hyperiondev

Hypothesis-driven debugging with the stack trace

Visit our website

Introduction

WELCOME TO THE TASK ON DEBUGGING YOUR CODE!

In this task, we will be learning about a hypothesis-driven process of debugging that can be implemented to make fixing code that does not behave as expected a more efficient experience .

Get in touch
Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!
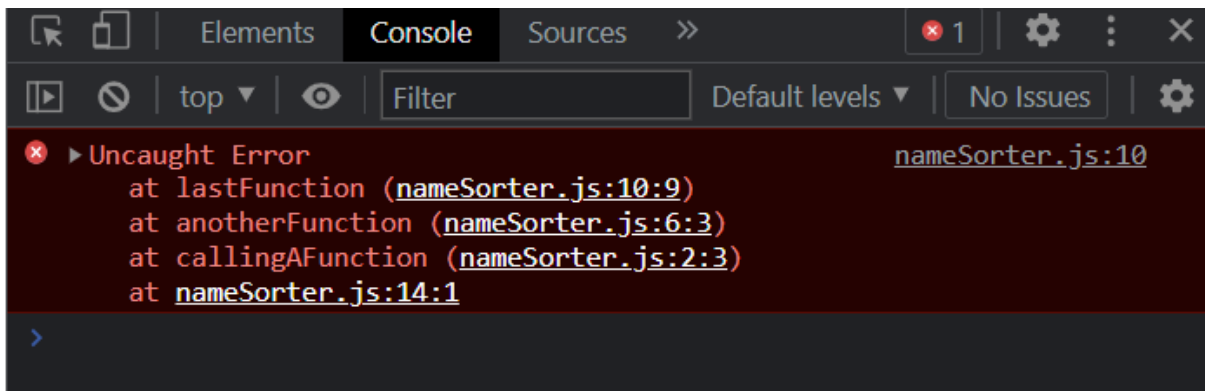
THE STACK TRACE

By now, you would have had the experience of writing some code and then when trying to execute that code, being faced with a bunch of red text in the console stating some error message with a whole lot of information that seems that might feel difficult to understand. Do not panic, this error report is what is called the **stack trace** in JavaScript! When you understand how to interpret the information in the stack trace, you will find it a great tool for helping you to debug your code.

A stack trace is a list of the functions, in order, that lead to a given point in a software program.

A stack trace is essentially a breadcrumb trail for your software.

Let us look at the following example where we are not using any real logic, but run through a series of function calls, to enable you to see what is represented in the stack trace when one of those functions throws an error:

```js
JS nameSorter.js > ...
1  function callingAFunction() {
2      anotherFunction();
3  }
4
5  function anotherFunction() {
6      lastFunction();
7  }
8
9  function lastFunction () {
10     throw new Error();
11 }
12
13 callingAFunction();
```

When we look at the stack trace above, we can derive some meaning by reading it from the top to the bottom.
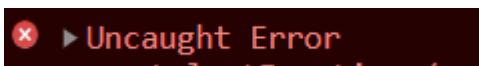

The first thing you will see is the type of error that has occurred. In this case, we do not have a specific error because we have thrown a generic error. Some general errors that you can encounter are **Reference errors** such as when a variable is not defined or **Syntax errors** such as "unexpected token" when a specific language construct was expected, but something else was encountered.

We can always use the **JavaScript error reference** to get a better idea of what type of problem we are facing so that we can narrow down what we are investigating.

After we identify what type of error we are facing, we can move from the top of the stack trace to the bottom. The last function call will be at the top of the stack and the first function call leading to that moment where the error occurred is at the bottom of the stack.

This means that lastFunction was called by anotherFunction and anotherFunction was called by callingAFunction. This is very helpful in tracing back through the code that you have written to determine what the code was meant to be used for, which narrows down even further what to look for as you are investigating.

When we look at each function call in the stack trace, we can also see which file, which line and at which character in that line the functions were called. In a very large code base with many modules, this is extremely useful when trying to find the problem. Imagine trying to figure out where an error is occurring without this information - you'd just be working blindly through code!
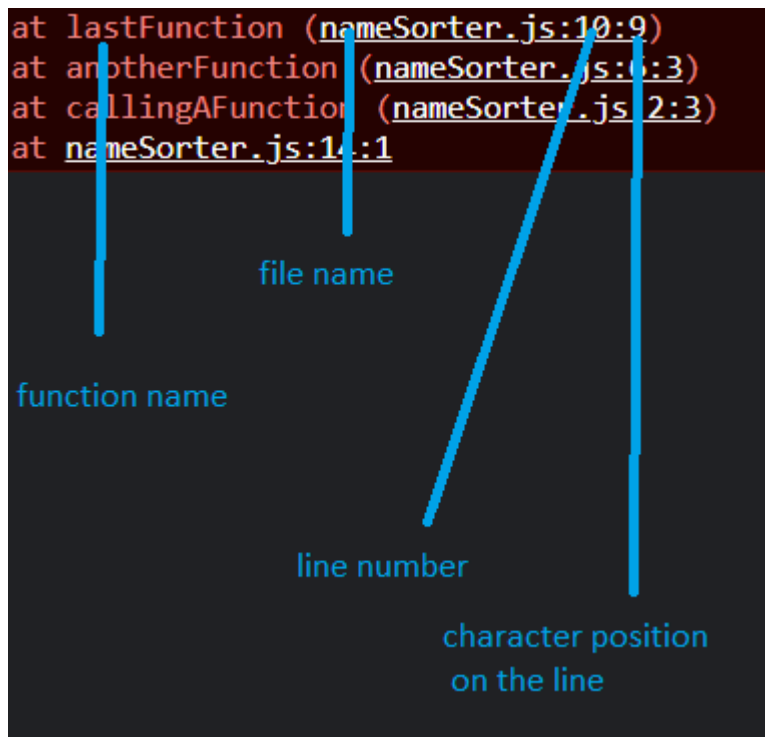
CREATING A HYPOTHESIS

Creating a hypothesis is the most important concept when debugging code. The more we know about the problem, the more targeted our bug fix attempts will be.

The following steps are a guideline for forming a hypothesis:

Step 1: Make observations.

Try replicating the issue that is being experienced. Record the event that triggers the issue and what the expected behaviour should be as well as what the unexpected behaviour is that you are experiencing. Unexpected behaviour could be errors or even incorrect outputs.

Other observations that you can record are where exactly the error occurs based on the stack trace, and which state your program was in when the error occurred. Looking at the code by following the stack trace also gives you a good idea of how the code is expected to behave.

Step 2: Question.

At this stage, one of the questions that you could ask is when the problem started. Is it something that has existed for some time or has the problem only arisen after a recent change? There are other questions that you could also ask based on the

particular problem depending on the issue at hand such as the user's role when the problem occurs, for example. Asking questions about the circumstances of the problem helps to be able to narrow down your search for the issue and form a better hypothesis.

Step 3: Hypothesis

Now that you are equipped with information, you can form a well-informed hypothesis as to where, when, and why the problem is occurring based on your understanding of the system as well.

Step 4: Make a prediction

Having formulated an idea of what might be the cause of the problem, make a prediction as to what will happen if you make specific changes and if those changes will or will not realistically fix the issue.

Step 5: Test your hypothesis

Now that you have hypothesised and made a prediction, you can make the necessary changes to test that hypothesis and prediction. If your hypothesis is incorrect, you can reevaluate and form a new hypothesis and related prediction based on any new information gathered, and employ deductive reasoning.

GAINING VISIBILITY INTO CODE

An important part of investigating code and forming a hypothesis is gaining visibility into exactly how lines of code are actually behaving vs how you may assume they behave. We can use the Debugger in the Chrome developer tools or in VS Code to do this. We strongly recommend looking at these 3 resources. You will be using Node.js later in the bootcamp, so you can always bookmark the resources about debugging Node.js with VS Code.

Debugging with Chrome Developer Tools

VS Code Debugger

VS Code Debugger Video

Another simple way to get visibility into how code is behaving is to use well-placed console logs to see the values of variables and whether those values correspond with what you assumed they should be.

A quick example of how console logs help us get visibility into the code is as follows. Say we have the following code for a function that takes a string input with colours separated by commas and spaces. Our function should split up the string into an array, count the colours and then output the count.

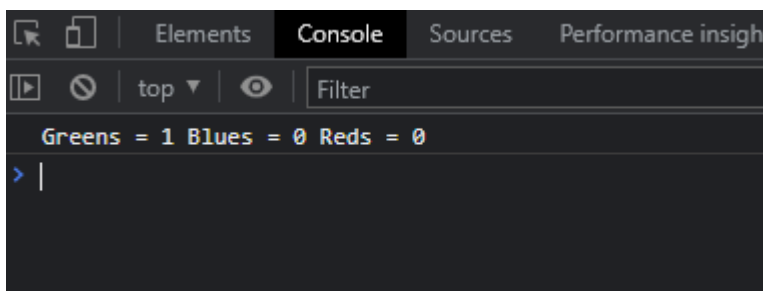Unfortunately, when we run the code we get the incorrect output:

```javascript
function countRedGreenBlue(inputColors) {
  let redCount = 0;
  let greenCount = 0;
  let blueCount = 0;

  splitColors = inputColors.split(",");

  splitColors.forEach((color) => {
    if (color === "red") {
      redCount++
    } else if (color === "green") {
      greenCount++
    } else if (color === "blue") {
      blueCount++
    }
  })

  return `Greens = ${greenCount} Blues = ${blueCount} Reds = ${redCount}`
}

console.log(countRedGreenBlue("green, red, green, green, blue, blue, blue"));
```

| Elements | Console | Sources | Performance insight |

top ▼ | Filter

Greens = 1 Blues = 0 Reds = 0

Oh no! Why is my output wrong?! Only one green has been counted! Never fear, we can output the values of variables to the console so that we can better understand how our code is behaving.

HyperionDev

Let's see what happens when we output the value of the array "splitColors" after we create it using the string.split() method

```
function countRedGreenBlue(inputColors) {
  let redCount = 0;
  let greenCount = 0;
  let blueCount = 0;

  splitColors = inputColors.split(",");
  //adding a console.log to see value of array
  console.log(splitColors);

  splitColors.forEach((color) => {
    if (color === "red") {
      redCount++
    } else if (color === "green") {
      greenCount++
    } else if (color === "blue") {
      blueCount++
    }
  })

  return `Greens = ${greenCount} Blues = ${blueCount} Reds = ${redCount}`
}

console.log(countRedGreenBlue("green, red, green, green, blue, blue, blue"));
```

What is the output in the console?

```
▶ (7) ['green', ' red', ' green', ' green', ' blue', ' blue', ' blue']
  Greens = 1 Blues = 0 Reds = 0
```

By looking at the strings in the array we can see that after splitting the input string, we still have whitespace at the beginning and end of some of the elements. We have not split correctly! Now that we know what the issue is, we can make a targeted change to the code i.e. changing the argument in the split method from "," to ", " or use .trim() thereby including the spaces in the splitting criteria and excluding them from the output. This removes the whitespace and our code compiles as expected.

```javascript
function countRedGreenBlue(inputColors) {
    let redCount = 0;
    let greenCount = 0;
    let blueCount = 0;

    splitColors = inputColors.split(",");

    splitColors.forEach((color) => {

        color = color.trim(); // trim to take away the white spacing

        if (color === "red") {
            redCount++
        } else if (color === "green") {
            greenCount++
        } else if (color === "blue") {
            blueCount++
        }
    })

    return `Greens = ${greenCount} Blues = ${blueCount} Reds = ${redCount}`
}

console.log(countRedGreenBlue("green, red, green, green, blue, blue, blue"));
```

Lets run the code again and we see that we now have the correct output!

```
Greens = 3 Blues = 3 Reds = 1
```

Some other handy methods to look into are the following:

- **console.debug** similar to console.log, but only visible in a debug console.
- **console.trace** outputs a stack trace to the console.

Compulsory Task 1

- Make a copy of the "Debugging" folder and rename it "CompulsoryTask". You will be making changes to the files in the "CompulsoryTask" folder and leaving the files in the "Debugging" folder as is.
- There are bugs in the code that need to be resolved. Debug the JavaScript code and ensure that you get the suggested output as outlined in the comments at the bottom of the JavaScript file.
- For each bug that you have resolved, please leave a comment identifying the change that you made.

Rate us
Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.