

# Exercise 07 – Debugging and Error Handling

## Objectives

To resolve error-ridden code and work with try, catch and error objects.

## Overview

In this exercise, there are a series of deliberate errors and problems. You are going to solve these issues on your journey to becoming JavaScript masters. Your instructor will walk you through Part 1 of the exercise to help you with understanding debugging and you will solve the remainder of the problems under your own steam.

Each of these exercises is commented out. As you attempt it, uncomment the relevant code and when you finish it comment it once again.


This exercise is scheduled to run for **40 mins (10 for each part)**.

## Exercise Set Up

1. In VSCode, open the file called **Errors.html** from the **Starter** folder inside **Ex07** of the **Exercises** folder.
2. If you have rebooted your computer or stopped *live-server* from running since the last exercise, then use VSCode's integrated terminal to navigate to the **Exercises** folder and run the command **live-server**.
3. Open your browser (if it doesn't automatically) and navigate to **http://localhost:8080** and follow the path to open **Ex07→Starter→Errors.html**.

## Part 1 – Debugging Function Calls

4. Remove the comments so that the code from Part 1 can be executed.
5. Take a second to review the code. You may notice errors, avoid the urge to fix them. Note down the errors you spot and what you think will happen.



6. Open the page in Chrome and bring up the developer console by pressing **F12**. Switch to the **Console tab**. There is an error message present. Check your assumptions from the previous step and explain what has happened.

7. Fix the code then test it in the browser, you should see no error messages.
8. Add the comments you removed around Part 1 before starting Part 2 of the exercise.

## Part 2 – Syntax Errors

9. As in Part 1, remove the comments from around the code block. Examine the code. This code has two syntax errors and one piece of bad practice to be resolved before the code will run.
10. What are the errors?

11. Fix them.
12. What is the bad practice?

13. Fix all errors in the code, including the bad practice.
14. After checking your code is working, re-apply the comments around Part 2.

## Part 3 – Using Throw

Throws are a very useful way of reporting back that something really bad has happened. Especially in larger JavaScript programs, they offer a way of reporting to functions what happened and where. This allows the program to either degrade gracefully or a developer to step in and solve issues.

15. Uncomment Part 3 of the exercise and take a look at the code.

Currently, we are trying to divide a number by a null. This will cause an error. Specifically, the operation will return the JavaScript value [Infinity](#).

We are going to apply defensive coding to ensure the operation does not execute when an incorrect value is offered.

16. Surround the line of code `kaboom= val / kaboom;` with an **if else** statement.

We are going to use the if statement to check if the operation will result in an error. By using this technique, we are checking whether we are about to cause an error – we only carry out the operation if it is safe.

17. Set the condition of the **if** statement to **isFinite(val / kaboom)** that when **true**, it returns a Boolean value that will let us know whether the error is going to happen.
- a) If the operation will not cause an issue, we can safely execute the assignment and then log out the value.

In many cases, we may be happy with this, but we are going to throw an error to report the problem.

18. What type of JavaScript error is most appropriate for the problem our code could cause?

19. Create a new instance of that object, setting an appropriate error message within the else statement and throw it, replacing **xxx** with the correct error object.

```
throw new xxx("Your operation causes a type exception");
```

20. Test the page in the browser and open the debug console. The error has not been handled, failing in the global scope as a result of the message you wrote in the previous step.

This is exactly what we want to see – our expression is going to fail and we have defended the code from executing it.

21. Comment out Part 3's code before you move onto Part 4.

## Part 4 – try, catch and function chains

In order to understand what happens when a function causes an error and we do not deal with it, we need to further explore the concepts from step 3. In most circumstances, we would expect a function to defend against incorrect code, and escalate via a throw, only when it cannot resolve the problem itself. When one function calls another or something odd happens inside, this is an essential technique for you to pick up as developers. It will ensure you and your colleagues do not spend fruitless hours searching for silent bugs!

22. Uncomment the code for Part 4 and examine it to ensure that you understand what it does.

We know from our work in Part 3 of this exercise that we will end up throwing and creating an error.

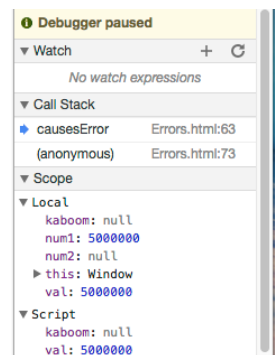
23. Open the page in Chrome and click the sources tab.

24. From the left hand pane select **Errors.html** and place a breakpoint against the code that calls the **causesError** function.



25. You will need to *refresh the page*. When you do this, you will pause at the function call. Step through the code (**F11** on Windows). This will move you line by line through the code. When you hit the if statement, stop.

Look to the right of the screen for the stack trace window. You will note that **causesError** has been called by the *global window object* (this is what the anonymous object refers to).



Our code is about to cause an error. Once that error occurs we need to resolve it in one of the calling functions. Currently, we are not going to do this to find out what happens next.

26. Walkthrough the code by pressing **F11** (or by clicking the appropriate icon) until the code fails.

**causesError** failed to handle the thrown error. Global failed to handle the thrown error. As a result, the program failed.

This is not ideal and we want to resolve the error before the program fails.

27. Wrap the function call in a **try...catch** block and **console.log** to report the **error name** and **message**.
28. Test the code again, working through the breakpoint again.

Now the error will escalate out of **causeError** (the function still fails). It will, however, be caught by the global object.

This shows we can catch an error anywhere in a call stack.

#### [Additional activities](#)

1. Move the try catch from Part 4 of the exercise inside the **causesError** function and note the differences.