



# Comprender los errores

## Curso de Fundamentos

### Introducción

Leer y comprender los mensajes de error es un requisito para los desarrolladores. A primera vista, muchos principiantes evitan los mensajes de error porque parecen "aterradores" y difíciles de entender porque incluyen términos con los que uno puede no estar familiarizado.

Sin embargo, los mensajes de error brindan a los desarrolladores un tesoro de conocimiento y le dicen todo lo que necesita saber sobre cómo resolverlos. Ser capaz de analizar los mensajes de error y las advertencias sin miedo le permitirá depurar eficazmente sus aplicaciones, recibir ayuda significativa de otros y empoderarse para seguir adelante cuando se enfrente a un error.

### Resumen de la lección

Esta sección contiene una descripción general de los temas que aprenderá en esta lección.

- Nombra al menos tres tipos de errores de JavaScript.
- Identifique dos partes de un mensaje de error que le ayuden a encontrar dónde se origina el error.
- Ser capaz de comprender cómo investigar y resolver errores.

### La anatomía de un error

Un error es un tipo de objeto integrado en el lenguaje JS, que consta de un nombre/tipo y un mensaje. Los errores contienen información crucial que puede ayudarle a localizar el código responsable del error, determinar por qué tiene este error y resolverlo.

Para todos los ejemplos de esta lección, debe ejecutar el código en la consola del navegador.

Supongamos que hemos escrito el siguiente código:

```
1 | const a = "Hello";  
2 | const b = "World";  
3 |  
4 | console.log(c);
```

Este código se ejecutará, pero generará un error. En términos técnicos, esto se llama "lanzar" un error. La primera parte de un error muestra el tipo de error. Esto proporciona la primera pista sobre a qué te enfrentas. Aprenderemos más sobre los diferentes tipos de error más adelante en la lección. En este ejemplo, tenemos un `ReferenceError`.



```
✖ Uncaught ReferenceError: c is not defined  
  at script.js:4
```

Se lanza un `ReferenceError` error cuando se hace referencia a una variable que no está declarada y/o inicializada dentro del ámbito actual. En nuestro caso, el mensaje de error explica que el error se ha producido porque `c is not defined`.

Los distintos errores de este tipo tienen mensajes diferentes según la causa `ReferenceError`. Por ejemplo, otro mensaje que puede aparecer es `ReferenceError: can't access lexical declaration 'X' before initialization`.

Como podemos ver, esto apunta a una razón completamente diferente a la original `ReferenceError` mencionada anteriormente. Comprender tanto el tipo de error como el mensaje de error es fundamental para comprender por qué recibe el error.

La siguiente parte de un error nos da el nombre del archivo en el que puede encontrar el error (en este caso, nuestro `script.js`), y también el número de línea.

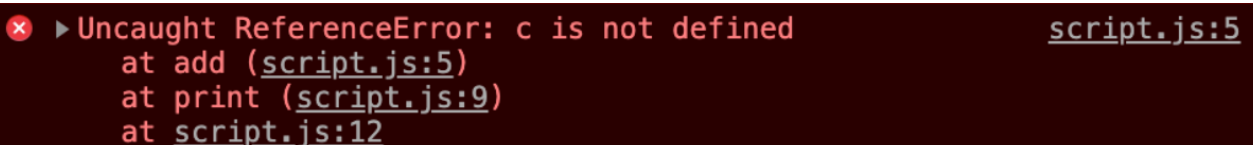
Esto le permite navegar fácilmente hasta la línea problemática en su código. Aquí, el error se origina en la cuarta línea de `script.js`, que se muestra como un enlace debajo del mensaje de error con el texto `at script.js:4`. Si hace clic en este enlace, la mayoría de los navegadores navegarán hasta la línea de código exacta y el resto de su script en la pestaña Fuentes de las Herramientas para desarrolladores.

A veces, la consola de tu navegador también mostrará la columna (o el carácter) en la línea en la que se produce el error. En nuestro ejemplo, sería `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:

```
1  const a = 5;
2  const b = 10;
3
4  function add() {
5      return c;
6  }
7
8  function print() {
9      add();
10 }
11
12 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:

```
1 | function helloWorld() {  
2 |     console.log "Hello World!"  
3 | }
```

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:

```
1 | const str1 = "Hello";  
2 | const str2 = "World!";  
3 | const message = str1.push(str2);
```

✖ ▶ Uncaught TypeError: str1.push is not a function [script.js:3](#)  
at [script.js:3:22](#)

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. Resuelve el problema de ["¿Qué salió mal? Solución de problemas de JavaScript"](#) . Asegúrate de descargar el código de inicio que contiene errores intencionales.

## Comprobación de conocimientos

Las siguientes preguntas son una oportunidad para reflexionar sobre temas clave de esta lección. Si no puede responder una pregunta, haga clic en ella para revisar el material, pero tenga en cuenta que no se espera que memorice o domine este conocimiento.

- [¿Cuáles son tres razones por las que podría ver un TypeError?](#)
- [¿Cuál es la diferencia clave entre un error y una advertencia?](#)
- [¿Cuál es un método que puedes utilizar para resolver un error?](#)

## Recursos adicionales

Esta sección contiene enlaces útiles a contenido relacionado. No es obligatoria, por lo que se la puede considerar complementaria.

- [Referencia de errores de JavaScript de MDN](#) .
- Lea el artículo de W3schools para encontrar [métodos de objetos de consola de ventana](#) adicionales .
- Además, mire la breve explicación en video de Steve Griffith [sobre la consola de herramientas de desarrollo de Chrome](#) .

 [Mejorar en GitHub](#)

 [Informar un problema](#)

[Ver registro de cambios de la lección](#)



Ver curso

[Marcar como completado](#)[Próxima lección](#)

## ¡Apóyanos!

El Proyecto Odin está financiado por la comunidad. ¡Únase a nosotros para ayudar a estudiantes de todo el mundo apoyando el Proyecto Odin!

[Más información](#)[Dona ahora](#)

THE ODIN PROJECT

Educación en codificación de alta calidad mantenida por una comunidad de código abierto.



### Sobre nosotros

[Acerca de](#)[Equipo](#)[Blog](#)[Casos de éxito](#)

### Apoyo

[Preguntas frecuentes](#)

### Guías

[Guías de la comunidad](#)[Guías de instalación](#)

### Legal

[Términos](#)[Privacidad](#)



[Contribuir](#)

[Contáctenos](#)

© 2025 El Proyecto Odin. Todos los derechos reservados.