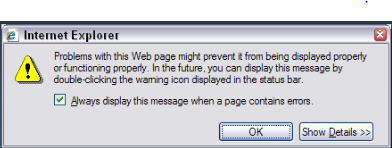
**What do we see when there is a JavaScript error on the web page:**

****

**Assumption in errors:**

* What if destination is null?

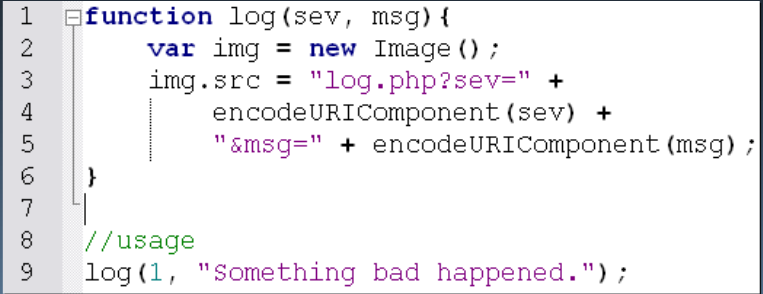
Asynchronous functions which take a callback often format the callback such that the first argument provided to the callback is the error, if any error is encountered, while the second argument is the successful retrieved value (if no error is encountered). That's exactly what's happening here. If destination or filename involved something which might throw an error, then the first argument you pass to cb could be the error.

* What if source is null?

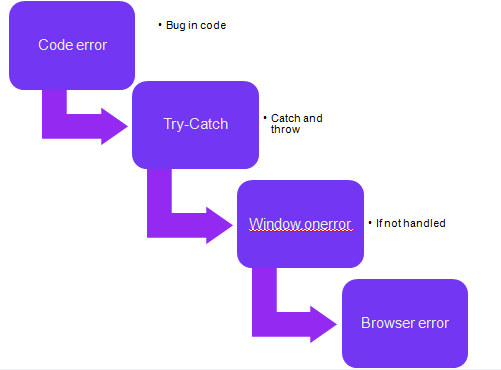
**Log Errors to the server:**

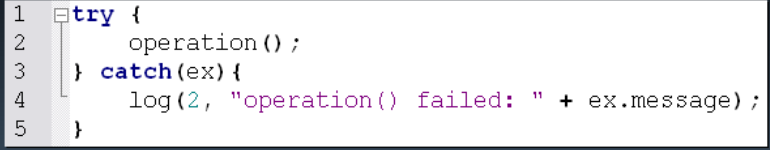
In both client-side JavaScript and Node.js, log data is handled by default via a global [console](https://developer.mozilla.org/en-US/docs/Web/API/Console) instance. While client-side JavaScript writes console data to the individual browser's developer console, Node.js console data is written to [stdout](https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)) and [stderr](https://en.wikipedia.org/wiki/Standard_streams#Standard_error_(stderr)). What this means, at least in terms of client-side JavaScript, is that log data is readily available to both the end user and the developer. While this can present some privacy issues, it can make debugging significantly easier in any environment. For those concerned about privacy, logging can also be disabled in production through logging frameworks or minification tools.

console.error(), as the name implies, outputs an error message. While the browser console will flag this as an error message, in Node.js applications the log message will be written directly to stdout.



* **Errors have to be handled by us not by browser**

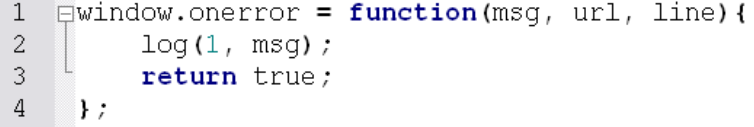




* Thrown errors contain extra information
* Errors that are caught are considered to have been handled

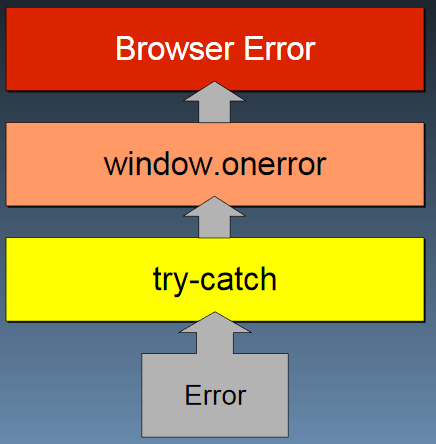
**Window.on Error:**

* Last stop before browser responds
* Return true to indicate not to respond
* Only supported in Internet Explorer and Firefox



**Error life Cycle:**

* **Indentify where errors might occur**



**Types of Errors:**

* Type coercion errors
* Data type errors
* Communication errors
* Invalid URL/post data
* Server response status
* No network connection
* Server response content

**Throw your own errors:**

**Throw or Try-Catch**

* Errors should be thrown in the low-level parts of the application–Utilities, core libraries, etc.
* Use try-catch blocks in higher level parts–Application-specific–Client-side business logic

**Distinguish between Fatal vs Non Fatal Errors:**

**Non Fatal Errors**

* Won't interfere with user's main tasks
* Affects only a portion of the page–Easily disabled/ignored
* Recovery is possible
* A repeat of the action may result in the appropriate result
* Don't tell the user it isn't working unless absolutely necessary

**Fatal Errors**

* The application absolutely cannot continue
* Significantly interferes with user's ability to be productive
* Other errors will occur if the application continues
* Message the user immediately!
* Reload

**Provide a DeBug Mode:**

* Assign a variable that is globally available
* try-catch should re-throw the error
* window.onerror should return false
* Allow the browser to handle the error

**Rules**

1. Assume your code will fail
2. Log errors to the server
3. You, not the browser, handle errors
4. Identify where errors might occur
5. Throw your own errors
6. Distinguish fatal versus non-fatal
7. Provide a debug mode