Error handling in JavaScript

When something goes wrong while a program is running, JavaScript uses the `try/catch` paradigm for handling those errors. Try/catch is fairly common, Python uses a similar mechanism.

First, an error is thrown

For example, let's say we try to access a property on an undefined variable. JavaScript will automatically "throw" an error.

```
const speed = car.speed
// The code crashes with the following error:
// "ReferenceError: car is not defined"
```

Trying and catching errors

By wrapping that code in a try/catch block, we can handle the case where `car` is not yet defined.

```
try {
  const speed = car.speed
} catch (err) {
  console.log(`An error was thrown: ${err}`)
  // the code cleanly logs:
  // "An error was thrown: ReferenceError: car is not defined"
}
```

Handling a new error object

When handling a thrown `Error` object, you must access the `message` property of the error object to display it correctly to the console.

```
let err = new Error('This is the error message');

try {
// computation
} catch (err) {
  console.log(`An error was thrown: ${err.message}`)
  // the code cleanly logs:
  // "An error was thrown: This is the error message"
}
```

Bugs vs Errors

Error handling via try/catch is not the same as debugging. Likewise, errors are not the same as bugs.

- Good code with no bugs can still produce errors that are gracefully handled
- Bugs are, by definition, bits of code that aren't working as intended

Debugging

"Debugging" a program is the process of going through your code to find where it is not behaving as expected. Debugging is a manual process performed by the developer. Sometimes developers use special software called a "debugger" to help them find bugs, but often they just use console.log() statements to figure out what's going on.

Examples of debugging:

- Adding a missing parameter to a function call
- Updating a broken URL that an HTTP call was trying to reach
- Fixing a date-picker component in an app that wasn't displaying properly

Error handling

"Error handling" is code that can handle expected edge cases in your program. Error handling is an automated process that we design into our production code to protect it from things like weak internet connections, bad user input, or bugs in other people's code that we have to interface with.

Examples of error handling:

- Using a try/catch block to detect an issue with user input
- Using a try/catch block to gracefully fail when no internet connection is available

In short, don't use try/catch to try to handle bugs

If your code has a bug, try/catch won't help you. You need to just go find the bug and fix it.

If something out of your control can produce issues in your code, you should use try/catch or other error-handling logic to deal with it.

For example, there could be a prompt in Jello for users to type in a new character name, but we don't want them to use punctuation. Validating their input and displaying an error message if something is wrong with it would be a form of "error handling".

async/await makes error handling easier

`try` and `catch` are the standard way to handle errors in JavaScript, the trouble is, the original Promise API with `.then` didn't allow us to make use of `try` and `catch` blocks.

Luckily, the `async` and `await` keywords do allow it, yet another reason to prefer the newer syntax.

.catch() callback on promises

The .catch() method works similarly to the .then() method, but it fires when a promise is rejected instead of resolved.

Example with .then and .catch callbacks:

```
fetchUser().then(function(user){
  console.log(`user fetched: ${user}`)
}).catch(function(err){
  console.log(`an error was thrown: ${err}`)
});
```

Example of awaiting a promise:

```
try {
  const user = await fetchUser()
  console.log(`user fetched: ${user}`)
} catch (err) {
  console.log(`an error was thrown: ${err}`)
}
```

As you can see, the `async/await` version looks just like normal `try/catch` JavaScript!