Catching Asynchronous Exceptions

This lesson describes how to handle asynchronous code exceptions.

WE'LL COVER THE FOLLOWING

- Asynchronous code
- Promises
- The await / async alternative

Asynchronous code

An asynchronous exception occurs when an asynchronous code throws an exception. Asynchronous code is often associated with the concept of `Promise` that is defined in the next section. In TypeScript (as well as JavaScript), asynchronous code is a piece of code that executes aside from the main thread of execution. The goal is to have a task running while another task is executed. For example, you can click buttons and write inside inputs while a response to a server is fetching information.

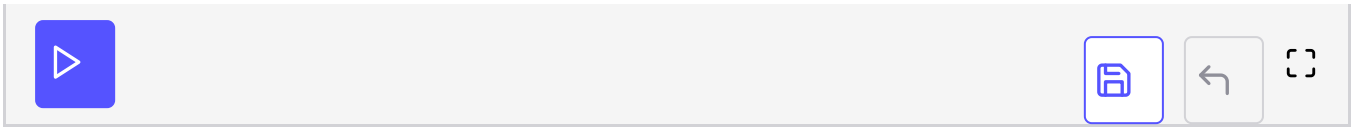
Promises

A `Promise` is the creation of asynchronous code. The mechanism allows calling the following function when a problem occurs or when there is a successful promise completed. In the following code, **line 3** throws an error. The problem is no code handles the error, hence it will get an unhandled exception.

> The following code executes unsuccessfully.

```
Promise.resolve("value to be in the argument of then")
    .then((response: string) => {
        throw new Error("Error message");
```

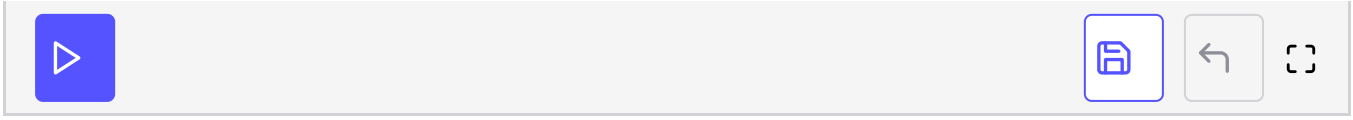
```
    });
```



Exceptions are tricky when mixed with the concept of `promise`. It is best to avoid having an exception with asynchronous code. However, exceptions happen. In the case of an exception, the promise that has the exception thrown must use the `catch` directly at the promise's level without wrapping the promise in a try-catch statement.

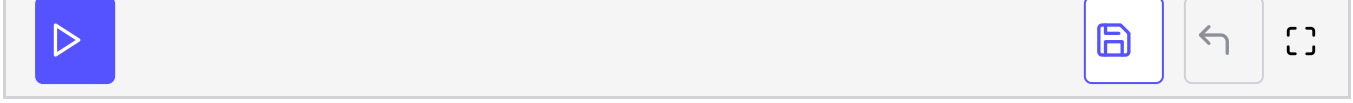
In the following code, **line 5** catches the error. It has the original error in `err`.

```
Promise.resolve("value to be in the argument of then")
    .then((response: string) => {
        throw new Error("Error message");
    })
    .catch((err: Error) => {
        console.log("Error Message#2", err.message);
    });
```



Promises don't have the concept of an explicit `finally` clause. To mimic a final clause, the code needs a `then` clause after the `catch`. In the asynchronous world jargon, a `finally` is a piece of code that is always executed regardless of the status of the execution (successful or unsuccessful). The following code at **line 14** is always called. The current example throws an exception. You can comment on **line 3** to see that **line 14** is still called.

```
Promise.resolve("value to be in the argument of then")
    .then((response: string) => {
        throw new Error("Error message");
        return "Test";
    })
    .then((response: string) => {
        console.log("Second then", response);
        return Promise.resolve(response);
    })
    .catch((err: Error) => {
        console.log("Error Message#2", err.message);
    })
    .then((response: string | void) => {
        console.log("Always called", response);
    });
```



In the code above, the “Error message” is written twice. First, because it is the error message that the console logs in the first catch. Then, the “Always called” is displayed because the last `then` is in the `finally` block.

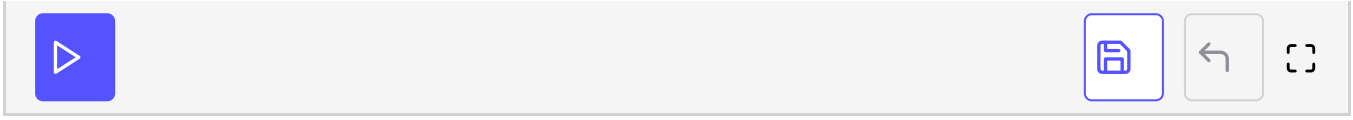
The await / async alternative

With ECMAScript 2017, asynchronous functions arrived to simplify the flow of code that was handled with promises.

The following code is a translation of the promise example with `await` and `async` to illustrate how to catch the exception.

```
function returnPromise(): Promise<string> {
  const p = Promise.resolve("value to be in the argument of then");
  throw new Error("Error Message");
  return p;
}

async function functionHandlePromise() {
  try {
    await returnPromise();
  }
  catch (err) {
    console.log("Error Message #2", err.message);
  }
  finally {
    console.log("Always called");
  }
}
functionHandlePromise();
```



The new syntax tries to accommodate developers and mitigates the confusion with several `then` and `catch` of promises. The modern and newer code base will lean toward the `async` and `await` model.