# Error Handling Best Practices

**Dan Wellman**
Senior Front-end Developer

# Following Error Handling Best Practices

- Improve the reliability and stability of applications by avoiding crashes
- Control failure in a way that minimizes the impact on users and system functionality
- Use graceful degradation to offer partial or alternative functionality
- Improve security by preventing malicious, unintended use of the application

# Error Handling in TypeScript

# Error Categories in TypeScript

- `InternalError` – an error occurred inside the JavaScript engine
- `RangeError` – you tried to access something out of the existing range
- `ReferenceError` – you tried to access a value that doesn't exist
- `SyntaxError` – you broke the syntactical rules of the language
- `TypeError` – you performed an illegal operation on a value
- `URIError` – an error encoding/decoding a URI

# The Purpose of TypeScript


- TypeScript prevents many common errors using the type system during development or compile time
- Most TypeErrors/ReferenceErrors/SyntaxErrors should be caught before a user would see them

# Testing

- A comprehensive testing strategy, including unit tests and integration/end-to-end tests
- Helps to catch errors that sneak through the development stage

# Handling Errors in TypeScript

- Handling errors is not about handling the errors that TypeScript will help avoid
- It means accounting for things beyond your control, like the availability of external resources

# Handling Errors in TypeScript

- Identify parts of your code where errors are likely to occur
- Take steps to handle these errors when they do occur

```
class Calculator {
  divide(dividend, divisor): number {
    return dividend / divisor;
  }
}
```

- As the parameters are supplied by the user, it may attempt to divide by zero
- It should take steps to avoid this by aborting the operation

Note: dividing by zero won't cause an actual error, it just returns `Infinity`

# Errors in TypeScript


**TypeScript gives you some basic tools for working with errors:**

- The `Error` type

```
const myError: Error = new Error();
```

- You use the `Error` constructor to create an object that conforms to the `Error` type

- Objects of the `Error` type will have the following properties:
  - `cause` (optional)
  - `message`
  - `name`
  - `stack` (optional)

```
const myError: TypeError =
  new TypeError();
```

- You also have built-in types for all the `Error` subtypes and constructors to create them

# Ensuring errors are explicitly typed promotes readability and type-safety.

# Errors in TypeScript


**TypeScript gives you some basic tools for working with errors:**

- The `Error` type
- The `throw` keyword

```
const myError: Error = new Error();
throw myError;
```

- **You can `throw` an error, which just means to raise it as an exception to be handled**

```
class Calculator {
  divide(dividend, divisor): number {
    if (divisor === 0) {
      throw new Error('Divide by zero');
    }
    return dividend / divisor;
  }
}
```

- **You can make the calculator `throw` an error when divide by zero occurs**

```
throw 'oops!';
```

- **You can `throw` any type of value**

**Always `throw` an actual error object to preserve information about the error.**

# Errors in TypeScript


**TypeScript gives you some basic tools for working with errors:**

- The `Error` type.
- The `throw` keyword.
- `try/catch/finally`.

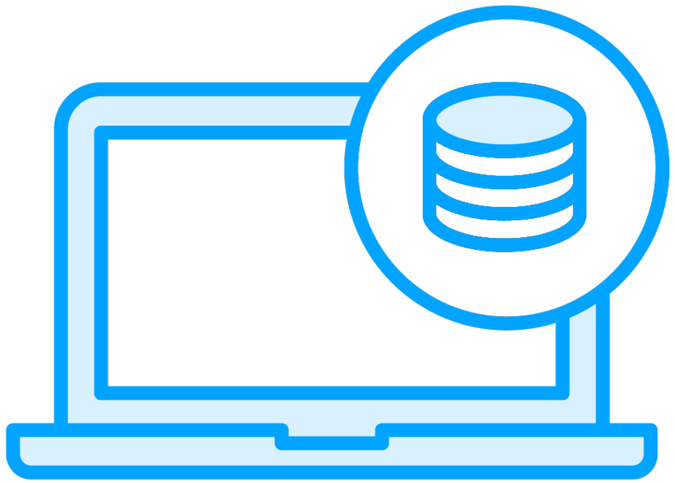
```
try {

  // something that might cause an
error

} catch (error) {

  // catch the “thrown” error and
  // handle it in some way

} finally {

  // do something after either
  // try or catch has finished

}
```

- You can wrap code that may fail in a `try` block
- If an error is thrown, it will be caught by the `catch` block, which receives the error
- `finally` executes after either (or both) `try` and `catch` have finished

Any errors not caught by you will bubble right up to the browser and potentially crash your application.

# Use `try`/`catch` Correctly

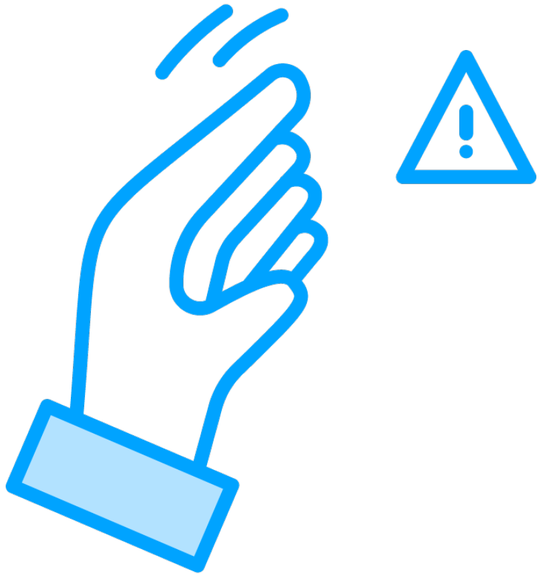
# LocalStorage Errors



- `LocalStorage` for the domain could already be full
- User may have disabled data storage
- These are realistic errors any app using `LocalStorage` should handle

# Never "swallow" errors, always do something with them.

# Handling Errors



- You should always take steps to mitigate the error or report it back to the user
- Users don’t like data loss!

# Never return a `string` instead of the error.

**Explicitly type the error as unknown for readability.**

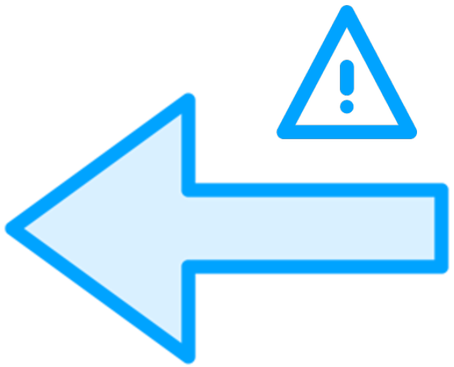
In a `try`/`catch`, you should always narrow the error down to a concrete type.

# Throwing vs. Returning

**Throw when the error cannot be recovered from and return when it can.**

# Use a union type including `Error` when a function returns an error.

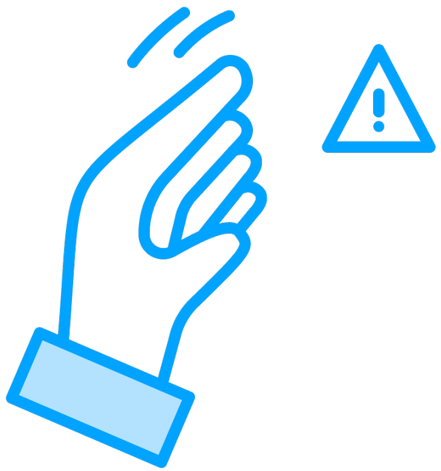
# Returning Errors



- Predictable control flow through the application
- Simplifies code by avoiding many instances of `try/catch`
- Avoids unhandled errors if you forget to catch them

# Logging errors is a critical part of any error-handling strategy.

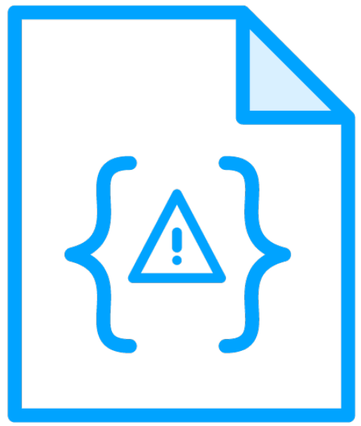
# Throwing Errors



- Clear separation between error code and "normal" code
- Reduced checks for whether something returned by a function is an error or not
- Errors can be caught at any level, without manually propagating through multiple functions

# Use Error Classes

# Benefits of Custom Error Classes



- Better identify the errors the application needs to handle
- Work with errors in a type-safe way
- Work with different errors in a consistent way
- Enhance errors with custom functionality

# Give Useful Errors and Error Documentation

# Document Errors

**Provide detailed information about each error including:**

- What caused the error to be thrown
- Useful name, description, error code
- Whether the error is thrown or returned

```
/**
 * @throws {StorageError} When LocalStorage is full or disabled
 */
```

**Use JSDoc's `@throw` tag to provide information on thrown errors**

# Document Errors

- Documenting all errors is useful for any developers working on the code
- You should also consider errors that need to be displayed to the user