* Errors, exceptions, warnings

Errors are caused when something goes wrong. They are usually the following:

System error: problem with the system or external devices

Programmer error: incorrect syntax or faulty logic

User error: user entered data incorrectly

Exception:

An error that produces a return value that can then be used by the program to deal with the error.

Stack trace:

An exception will produce a stack trace, a sequence of functions that lead to the point where the error occurred.

Warnings:

A warning can occur if there’s an error in the code that isn’t enough to cause the program to crash.

* The importance of testing and debugging

It is good to fail loudly during development to fix those errors early. It is then good to fail gracefully, so that it doesn’t affect the user experience.

* Strict mode

A mode that produces more exceptions and warnings. Prohibits the use of some deprecated features.

Linting tools:

Can be used to test the quality of javascript code, beyond using strict mode.

Feature Detection:

Used to check if browser support for a feature exists before calling the method.

* Debugging in the browser

The process of finding bugs

The trusty alert:

The most basic form of debugging is to use alert()

Using the console:

Another way to debug is to print out to the console with console.log().

Debugging tools:

Most modern browsers have debugging tools that let you set breakpoints that will make it pause at certain points. One of the most useful commands is the debugger keyword.

* Error objects

An error object can be created by the host environment when an exception occurs, or it can be created using a constructor function, like this:

*const error = new Error();*

The contructor takes a parameter that is the error message. There are a variety of error objects.

* Throwing exceptions

Previously, those errors are thrown automatically by javascript, but we can also throw our own exceptions using the throw statement. It is best practice to specifically throw an error object, like this:

*throw new Error('Something has gone badly wrong!');*

* Exception handling

When an exception occurs, the program terminates with an error message. This will make it appear as if the program crashed if it occurs in production.

You can handle exceptions by catching the error.

If we suspect a piece of code may result in an exception, we can wrap it in a try block. This will run the code inside the block as normal, but if an exception occurs it will pass the error object that is thrown onto a catch block. Here is an example:

*function imaginarySquareRoot(number) {*

*'use strict';*

*try {*

*return String(squareRoot(number));*

*} catch(error) {*

*return squareRoot(-number)+'i';*

*}*

*}*

A finally block can be added after a catch block. This will always be executed after the try or catch block, regardless of whether an exception occurred or not. It is useful if you want some code to run in both cases. We can use this to modify the imaginarySquareRoot() function so that it adds + or – to the answer before returning it:

*function imaginarySquareRoot(number) {*

*'use strict';*

*let answer;*

*try {*

*answer = String(squareRoot(number));*

*} catch(error) {*

*answer = squareRoot(-number)+"i";*

*} finally {*

*return `+ or - ${answer}`;*

*}*

*}*

* Tests

Tests are important.

They can be carried out with a function to test if it works right.

Test driven development (TDD):

The process of writing tests before any actual code. The work flow could look like this (red-green-refactor):

1. Write tests (that initially fail)
2. Write code to pass the tests
3. Refactor the code
4. Test refactored code
5. Write more tests for new features

Testing frameworks:

Testing frameworks provide a structure to write meaningful tests and then run them. One of them is Jest, a TDD framework created by Facebook.