### 1 – Exceptions

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

unicorn();

<< ReferenceError: unicorn is not defined

### Warnings

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

pi = 3.142;

<< JavaScript Warning: assignment to undeclared variable

## 2- **Strict Mode. (very interesting concept)**

Increasing the chance of errors might seem like a bad idea at first, but it’s much better to spot errors early on, rather than have them cause problems later.

Ex1:

(function() {

'use strict';

// All your code would go inside this function

}());

Ex2:

function strictly(){

'use strict';

// function code goes here

## 3 - **Debugging in the Browser**

### The Trusty Alert

function amIOldEnough(age){

if (age = 12) {

alert(age);

return 'No, sorry.';

} else if (age < 18) {

return 'Only if you are accompanied by an adult.';

}

else {

return 'Yep, come on in!';

}

}

The alert method inside the if block will allow us to see the value of the age variable at that point. Once we click on OK, we can then check the function returns the correct message.

### Using the Console

function amIOldEnough(age){

console.log(age);

if (age < 12) {

console.log(`In the if with ${age}`);

return 'No, sorry.';

} else if (age < 18) {

console.log(`In the else-if with ${age}`);

return 'Only if you are accompanied by an adult.';

} else {

console.log(`In the else with ${age}`);

return 'Yep, come on in!';

}

}

## **Error Objects**

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

const error = new Error();

This constructor function takes a parameter that’s used as the error message:

const error = new Error('Oops, something went wrong');

* TypeError is thrown when there’s an error in the type of value used; for example, a string is used when a number is expected.
* URIError is thrown when there’s a problem encoding or decoding the URI.
* InternalError is a non-standard error that is thrown when an error occurs in the JavaScript engine. A common cause of this too much recursion.

### Crunching Some Numbers. –

### Important: I am confused about it. I’ll go back and restudy again.

To demonstrate the TDD process, we’ll have a go at creating a small library called 'Number Cruncher' that will contain some functions that operate on numbers. The first function we’ll try to implement will be called factorsOf() . This will take a number as a parameter and return all the factors (The factors, or divisors, of a number are any integers that divide exactly into the number without leaving a remainder. For example, the factors of 6 are 1, 2, 3 and 6.) of that number as an array.

Since we’re doing TDD, we need to start by writing the tests first, so create a file called numberCruncher.test.js and add the following code:

test('factors of 12', () => {

expect(factorsOf(12)).toEqual([1,2,3,4,6,12]);

});

#### **Use of toEqual()**

We have used the toEqual() match in this test. This is because we are testing an array.

This test says our factorsOf() function should return an array containing all the factors of 12 in order, when 12 is provided as an argument. If we run this test, we can see that it fails spectacularly:

jest -c {}

<< FAIL ./numberCruncher.test.js

● factors of 12

ReferenceError: factorsOf is not defined

at Object.<anonymous>.test (numberCruncher.test.js:2:10)

at process.\_tickCallback (internal/process/next\_tick.js:103:7)

✕ factors of 12 (6ms)

Test Suites: 1 failed, 1 total

Tests: 1 failed, 1 total

Snapshots: 0 total

Time: 1.424s

Well, what did you expect? We haven't written any code yet! Let’s have a go at writing the factorsOf() function. Add the following to the top of the numberCruncher.test.js file:

'use strict';

function factorsOf(n) {

const factors = [];

for (let i=1; i < n ; i++) {

if (n/i === Math.floor(n/i)){

factors.push(i);

}

}

return factors;

}

This function creates a local variable called factors and initializes it as an empty array. It then loops through every integer value from 1 up to n (the number that was given as an argument) and adds it to the array of factors using the push() method, if it’s a factor. We test if it’s a factor by seeing if the answer leaves a whole number with no remainder when n is divided by the integer 1 (the definition of a factor).

#### **This Isn't Totally Realistic**

To make things easier in this example, we're putting the code into the same file as the tests, but in reality you'd usually keep them in separate files.

Try running the test again:

jest -c {}

<< FAIL ./numberCruncher.test.js

● factors of 12

expect(received).toBe(expected)

Expected value to be (using ===):

[1, 2, 3, 4, 6, 12]

Received:

[1, 2, 3, 4, 6]

Difference:

- Expected

+ Received

@@ -2,7 +2,6 @@

1,

2,

3,

4,

6,

- 12,

]

at Object.<anonymous>.test (numberCruncher.test.js:14:25)

at process.\_tickCallback (internal/process/next\_tick.js:103:7)

✕ factors of 12 (12ms)

Test Suites: 1 failed, 1 total

Tests: 1 failed, 1 total

Snapshots: 0 total

Time: 0.801s, estimated 1s

Ran all test suites.