Session 5

Recap

In the previous session we covered the following:

- Functions
- High Order Functions
- Objects (Revisited)
- Destructuring

Mutation

When you mutate you copy everything about it. The same principle applies to when we program and we mutate variables (objects, data structures etc).

... in Arrays

You should not mutate anything when code, as you're changing the context of a variable that you recieve which may have side effects on the logic or the flow of control in your code.

Let's look at mutation in arrays:

```
const numbers = [1, 2, 3, 4 ,5];

// let's try copying the numbers

const copiedNumbers = numbers;

console.log('Copied numbers', copiedNumbers);
```

Great, we have copied the numbers across into a new array. But wait. What are arrays? They are reference types. When we create an array, JavaScript creates a reference to that array in memory (which basically is a reference to where it lives in memory). The space granted in memory to this array is dynamic. As we have *copied* the `numbers` into `copiedNumbers` we have essentially copied the *reference* of the first array into the second, meaning, if we change `numbers`, it'll also change `copiedNumbers` 😱 :

```
const numbers = [1, 2, 3, 4 ,5];

// let's try copying the numbers

const copiedNumbers = numbers;

console.log('Copied numbers', copiedNumbers);

copiedNumbers.push(10);

console.log('Numbers', numbers, 'Copied numbers', copiedNumbers);
```

This will give us:

```
Numbers [1, 2, 3, 4, 5, 10] Copied numbers [1, 2, 3, 4, 5, 10]
```

`10` appears in both, but we only added it to `copiedNumbers`. Arghh! Imagine if this was an array containing a set of banking transactions. If we added another transaction to our 'copied' array we'd end up referencing an incorrect data set when what we really want is to create a `new` array.

... in Objects

Let's look at how mutation would be when dealing with objects:

```
const bankAccount = {
    name: 'John Doe',
    accountNumber: '13213123',
    transactions: []
};

const copiedBankAccount = bankAccount;

copiedBankAccount.balance = -100;

console.log('Bank Account', bankAccount, 'Copied Account', copiedBankAccount);
```

The balance has been copied in **both** objects. We only wanted to add it to the *copied* version - d'oh

If we illustrated the transactions example:

```
const bankAccount = {
    name: 'John Doe',
    accountNumber: '13213123',
    transactions: []
};

const copiedBankAccount = bankAccount;

copiedBankAccount.balance = -100;
copiedBankAccount.transactions.push({
    id: 1,
    ts: 123453532523,
    from: 'me',
    to: 'you',
    amount: 100
})

console.log('Bank Account', bankAccount, 'Copied', copiedBankAccount);
```

The *transactions* have also been replicated in **both** objects. In the real world of course, the variable wont be called `copiedBankAccount`, so you can see if if was just called `savingsBankAccount` that any manipulation could mislead you to use this object without realising it is not a **true**, detached copy.

Spread Operator

So we have seen that what we deem copying in JavaScript is not truly a clean copy, it's merely a reference copy.

Thankfully, JavaScript introduced a feature which is now heavily used in ES6, called the *spread operator*. Prior to this introduction there were ways to copy arrays and objects but those approaches were cumbersome.

The *spread operator* allows us to cleanly copy arrays and objects in a *detached* manner.

The *spread operator* uses the three prefixed dots notation (`...foo`), which you could say looks identical to the use of a *rest operator*. The key difference to note is that the *spread operator* **expands** contents where as the *rest operator* you could say **condenses** them.

... with Arrays

To use the *spread operator* with Arrays let's look at this example:

```
const names = ['John', 'Jill', 'Sue'];

const copyOfNames = [ ...names ];
```

Here, the contents of `names` are expanded into a new array as an initialised value to the `copyOfNames` variable. So if we now try to amend the `copyOfNames` variable, it will not have any effect on the original, source array.

... with Objects

You can also expand, copy an object:

```
const album = {
    name: 'foo',
    song: 'bar'
}

const copiedAlbum = { ...album };
console.log(copiedAlbum);
```

This can be futher expanded to adding other properties:

```
const album = {
    name: 'foo',
    song: 'bar',
    hobbies: ['football'],
    lyrics: {
        numLines: 100,
        verses: 20,
        lines: ['hello', 'bye']
    }
}

const copiedAlbum = { ...album };
copiedAlbum.name = 'Bob'
```

```
console.log('Album', album, 'Copied Album', copiedAlbum)
```

For mroe reading about the *spread operator*, refer to this link.

Error Handling

Not everything goes according to plan. In programming it's no different and as such, we need to handle such scenarios.

In the programming world we will have scenarios that can present errors and how we handle them is important both from an application point of view and the other from an end user's point of view.

Let's look at the following:

```
const divide = (a, b) => a / b;

console.log(divide(10, 5)); // we get 2

console.log(divide(10, 0)); // oh no!
```

So how do we handle such scenarios?

Throwing errors

A possible approach taken by developers in most languages is to *throw an error* which basically means that the code you're invoking, hasn't done what it has expected or an argument you provide to a function may not be in the form it needs to be, therefore, an error is returned by the invoked logic.

This is also possible in JavaScript and as a developer, you can decide whether it is something you would like your application to do. For example let's look at this code:

```
const divide = (a, b) => a / b;

console.log(divide(10, 5)); // we get 2

console.log(divide(10, 0)); // oh no!
```

When we look at testing, we would after running tests find that we need to handle such scenario where we are provided a zero to divide by. We can gracefully handle this by simply returning a console message or a null value (the latter may seem strange to do), but what is better, is to throw an error:

```
const divide = (a, b) => {
    if (b === 0) {
        throw new Error('Unable to divide by zero');
    }

    return a / b;
};
```

```
console.log(divide(10, 0)); // Unable to divide by zero
```

At least by throwing an error, we can return an error for the caller of our function to handle the code locally as well as give a human readable reason why it failed to process the arguments.

Best Practice for error messages

As with coding logic, error messages also have some traits that you should follow in order to provide clear and concise outcomes:

- Assume your code will fail, so handle or provide an outcome that external users can handle (i.e. throw an error)
- Log errors where appropriate so it's easier to understand a potential reason for the failure
- If rendering an error message back to the user, be clear
- An error is something unexpected. Not an invalid password or invalid username. These are expected outcomes of your application. You would never tell the user if their password was incorrect or their username, the message should be vague (otherwise you're giving malicious users an idea).

Try/Catch

So what happens to the code that calls our `divide` function? How can we *gracefully* handle the error? Well we can use a `try` and `catch` block:

```
const divide = (a, b) => {
    if (b === 0) {
        throw new Error('Unable to divide by zero');
    }

    return a / b;
};

try {
    const result = divide(10, 0);
} catch (e) {
    // handle error here:
    console.error(e);
    alert(`Sorry you cannot divide by zero, try again`);
}
```

The example demonstrated is a simple one but the concept of how we can handle errors is the same process in all scenarios. So why would we need to handle errors?

- You may be using a third party library which may cause errors to be thrown because you have provided it parameters that are not valid
- We may be recieving data from an external server, or a service and we cannot guarantee that the the data we get will be the same all the time, i.e it might be missing key areas.
- Not everything in the world works perfectly now does it

Testing

Why not just put our into Production and let the users test it. Yes why ..... not!

There's numerous examples of pitfalls organisations have fallen in simply because they have not performed the means of testing their applications.

There are many forms of testing and I'll explain them just for your sanity but we will be covering in this session how to write tests in JavaScript.

But why do we need to testing?

- Goal: Tests check whether an application behaves as we expect it to.
- Tests safeguard against unwanted behaviour when changes are made.
- When we add tests, we in turn add automation, and thus this becomes efficient in the long term.

When it comes to testing, there are multiple forms of testing.

Functional Testing

- Manual Testing - Been there, done that ...Zzzz
- Integration Testing - This is where you as a developer write tests to verify that any external parts you rely on work with your program.
- Smoke Testing - When testers receive code they'll usually run a set of tests to verify that your latest changes won't break their tests.

Non-Functional Testing

- Performance Testing - Is the system performing and meeting the KPIs?
- Load Testing - Can the application handle millions of requests without degrading the performance.
- Stress Testing - Can your application work outside of it's boundaries? Your program handles only 100 request, what if I sent it 200?

These different stages or levels of testing will be covered in the future with your organisation or future bootcamps.

Unit Testing

One area we will focus on are *unit* tests.

When we mention the word unit, we simply mean testing a small piece of functionality (which is usually a function or a piece of logic that performs an action). Several of these running together is referred to as a **suite** of tests.

Testing in JavaScript

In order to write tests in JavaScript, we need an environment/framework that allows us to (a) write the tests and (b) run the tests to actually target the application code.

One testing framework that is highly popular right now is `Jest`.

> There are others such as `Mocha` and `Jasmine`, `Karma` out there but we'll be using `Jest` for all our testing in this bootcamp. Feel free to look into the others as extra reading 📚.

Getting Started

Before we start we must ensure we have Node JS installed. You can download and install it from here.

Once installed you can verify your node installation using the following command at a terminal/command prompt:

```
npm -v
```

If this renders a version number then Node is installed and ready to use. If not, re-try the installation.

In your lab session folder, you will find a `package.json` file. We can now install dependencies we require to start writing tests:

At a terminal/command prompt:

```
npm install
```

The `@types/jest` dependency allows us to find special `jest` matchers within our IDE when using intellisense (we'll look at those later).

What this does is install `Jest` as a dev dependency (using the `--save-dev` switch - this can also be abbreivated to `-D`).

Once the installation finishes, you will now notice a few things:

1. A new `node_modules` folder in the root which contains all of your dependencies. We DO NOT include this when pushing to Git (as we add it to our .gitignore).
2. Inside `package.json` you will a new `devDependencies` section that now shows `jest` and it's version.

 If you download a project that has a `package.json` file, then you need to run `npm install` which will then download all dependencies for that project. This is why we don't add `node_modules` (firstly it's way too big to upload and secondly, versions change so we want to ensure we use the latest version or the one the application needs in order to function).

Running Tests

Now that we have set up Jest, we can now run tests. To run tests you can type the following at a command/terminal prompt:

```
npm test
```

or the abbreviated form, `npm t`

You will get an error stating that we need to have at least one test.

Creating Test Files

In general whenever we write some code, we must accompany it with a test file. The convention is to add either of the following suffixes to your test files: `.test.js` or `.spec.js`.

Test frameworks such as `Jest` will look for those extensions by default.

If you say create a file called `Messages.js`, then you must create a test file for it with the same name but with either the `.test.js` or `.spec.js` extension (i.e. `Messages.test.js`). Try to stick to **one** extension convention. You will note this when you look at the code used within your organisation.

> The location of the test file again is down to how your organisation arranges their code. Some like to create a separate `test` folder and mirror the structure whereas others (myself included), prefer to place the test file in the same location of the implementation code.

> People also refer to test files as test specifications. Why? because in theory your tests are the documentation for your application. They specify what your code's intentions are.

AAA Pattern

> Arrange, Act, Assert Pattern

This is the art to unit testing.

Unit testing is where we write individual tests for each unit/feature of work we produce.

- ARRANGE - This is where you perform set up tests in preparation for you to run a test. This could be merely initialising some variable to actually pre-populating, say an array to pass to your method that you want to test.
- ACT - This is the actual call your feature. The ACT, you can think of as the "action". What "Action" do I need to perform to test my method? ... Call the method!
- ASSERT - This is where you look at the result of running your feature. Is it the value you expected?

Anatomy of a test

So how does a test look like when using `Jest`. Let's create a file called `Sample.test.js` inside a `testing` folder in `session-6`.

To add a test, we can use several ways to define a single test case. We can simply just use the `test` function or we can use the `it` function. They both do the same thing but when grouping tests together the latter is usually preferred.

For example, this is how we can define a test inside the `Sample.test.js` file:

```
test('This is a simple test', () => {
  // logic for your test code
});
```

Alternatively we can use `it`:

```
it('This is another simple test', () => {
  // logic for your test code
});
```

Either approach depends on the next section when we look at test suites or grouped tests. If you use a group, it's preferred to use the `it` form. Both `test` and `it` are used as they read naturally when you read the test results and the test themselves, i.e.:

> *it* should return X when passed a value of Y *test* should return bank details when account number provided ...

If we want to group tests together, which is a good thing, we can use the `describe` function/block. With the `describe` block, we tend to use `it` blocks inside:

```
describe('MessageSender:', () => {
  it('should send message in English', () => {
    // logic
  });

  it('should send message in French', () => {
    // logic for french
  });
});
```

When we run the grouped tests, we will see a nested output all under 'MessageSender:'.

This will then render outputs that look like this on a test report:

```
MessageSender:
 - should send message in English
 - should send message in French
```

By having grouped tests, it allows you to organise tests that **relate** to one another or fall under the same area of concern, for example all tests to do with registering an account, another to log a person out.

We'll look at other aspects of `jest` and what features we can utilise to write good tests as we progress further.

Writing our first test

Inside the session-5 lab folder, create a new file `Calculator.js` and then also a test file for it, `Calculator.test.js`.

It should look like this:

Whenever want to write a test, we start by adding a `test` block that takes a string as it's first argument and an anonymous function as the second. This is where you put the test code.

So inside `Calculator.test.js` add this:

```
test('Our first test', () => {
  // blank
});
```

Now run the test: `npm test`

Our test is passing but we havent put any code in place? `Jest` is just interested in running the function and is looking for code that passes. Here we have none so by default it will pass - be careful with this as you may forget to put actual test code inside and assume tests are passing 😬

Let's make our tests `fail`:

```
test('Our first test', () => {
  throw Error('Should fail test');
});
```

Now run the test: `npm test`

 We have a failing test!

Ok, let's empty our `Calculator.test.js` file and add some code that we can test.

Let's add some code and now check whether we are getting what we expect:

Inside `Calculator.js` add a function called `add`:

```
const add = (num1, num2) => {}
```

Now open the `Calculator.test.js` file and add a test case for it:

```
test('When given two numbers, their sum is returned', () => {
  const result = add(1, 1);
})
```

You will see that `jest` will complain that `add` is not defined. Lets import it into our test file:

```
import { add } from './Calculator'; // no JS extension needed when using Node

test('When given two numbers, their sum is returned', () => {
  const result = add(1, 1);
})
```

We'll still get an error - why? We haven't `exported` the `add` function.

```
export const add = (num1, num2) => {}
```

Re-run the code and our test will pass. But hold on! We haven't actually verified the `result` is actually the sum?

`Jest` provides a way for us to verify the content of variables, using the `expect` function. The `expect` function takes the variable we are looking to verify and then also provides us with chained functions that we can use for the verification:

```
import { add } from './Calculator'; // no JS extension needed when using Node

test('When given two numbers, their sum is returned', () => {
  // Arrange:
  const num1 = 1;
  const num2 = 1;
```

```
  // Act:
  const result = add(num1, num2);

  // Assert:
  expect(result).toBe(2); // the 'toBe' is an assertion function
})
```

Here you can see we used an `expectation` using the `expect` function and then used one of the many `matchers` jest provides in the form of `toBe`.

What you may also have noticed is how it naturally reads like English:

> *Expect the result to be 2* This is why some people may say that our tests act as a documentation for our implementation.

To practice testing, we will go through several katas. Don't worry about not understanding testing right away, it does take time and practice.

Challenge: Sum of sequence

Given you have a `begin`, `end` and `step` set of arguments. Your task is to write a function that takes these non-negative numbers and returns the sum of integers from `begin` to `end` that increase by the number of `step`.

If `begin` is greater than `end`, the function should return `0`.

Examples:

```
sumOfSequence(1, 4, 1) // returns 10 because 1 + 2 + 3 + 4
sumOfSequence(2, 6, 2) // returns 12 because 2 + 4 + 6
```

Challenge: Bouncing Ball

A child is playing with a ball on the `nth` floor of a tall building. The height `h` of this floor is known.

He drops the ball out of the window. The ball bounces to two thirds of it's height, i.e. 0.66.

His mother looks on from a window that is 1.5m from the ground.

How many times will the mother see the ball pass her window (including the time the ball falls and whilst its bouncing)?

Assume you have the following arguments to your function:

`h` is a float that is greater than `0` `bounce` is a float that is greater than `0` and less than `1` `window` must be less than `h`

If all of the above argument criterias are matched, return the outcome of occurences, as an integer, otherwise return `-1`

Challenge: Amount as Coins

Given you have available coins as 25, 10, 5, 2 and 1, write a program to work convert an amount of money to coins.

```
For example: If the amount is 46
Output: 25, 10, 10, 1
```

You are expected to write unit tests

Challenge: Number of Boomerangs

A boomerang is a V-shaped sequence that is either upright or upside down. Specifically, a boomerang can be defined as a sub-array of length **3**, with the first and last digits being the same and the middle digit being different.

For example:

[3, 7, 3], [1, -1, 1], [2, 9, 2] are all boomerangs

Note: A continuous sequence is NOT a boomerang such as [5, 5, 5].

Create a function that returns the total number of boomerangs in an array of numbers

Example:

```
[3, 7, 3, 2, 1, 5, 1, 2, 2, -2, 2]
// 3 boomerangs in this sequence: [3, 7, 3], [1, 5, 1] and [2, -2, 2]
```

Caution, boomerangs can overlap:

```
[1, 7, 1, 7, 1, 7, 1]
// 5 boomerangs: [1, 7, 1], [7, 1, 7], [1, 7, 1], [7, 1, 7, [1, 7, 1]
```

Challenge: People Operations

Given you have been provided a database of users, write functions that:

- Return all males.
- Return all females.
- Returns all people under (and including) the age of 40.
- Returns all people with their first name abbreviated to an initial, followed by a dot, a space and then their surname. The title should be prefixed as Mr or Ms based on gender.
- Return the average age amonsgt the group of people.
- Returns all people who have 'Finnish' as their subject.
- Return all people who's surname begins with 'M'.
- Group all people by their respective subject.

You are expected to write tests for each of these and use the same test data.