Functional JS

Introduction

- Functional Programming (FP) favours:
  - functions over objects
  - transformations over modifications
  - immutable over mutable data

- FP is not in competition with OOP; it is often used alongside it

- FP is typically applied when you're seeking to extract new information from existing information

Higher Order Functions

- A higher order function is one that accepts a function as an argument and/or returns a function

  ```
  element.addEventListener(
    'click',
    function(event) { ... }
  );
  ```

- A decorator is a higher order function that accepts a function and returns a modified version of it

Pure Functions

- A pure function is one that always yields the same output given the same input; it produces no side-effects

- The function below - pure or impure?

```
const stripAndCapitalise = str => {
  return str.trim().toUpperCase();
}
```

- Pure functions are easier to maintain and test than those dependent on external state

Immutability

- Why favour immutable objects? There are lots of reasons but in a multi-component app the sharing of mutable objects between components can be painful; a change in one component can affect n others in unpredictable ways

- JS doesn't make it easy to make objects immutable but there are some options:
  - Class with no setters (not really immutable)
  - Object.freeze (shallow)
  - Make a copy using spread (shallow)
  - JSON.stringify and JSON.parse (incomplete)
  - 3rd party lib, e.g. Immutable.js (more to learn)

- The alternative is to treat your objects as if they were immutable and make copies/favour methods that transform the data

Currying

- Put simply, currying is the process of creating new functions from an existing one

- A function with n unknowns can be used to create one or more new functions with fewer unknowns, e.g.:

```
str.substr(0, 1);
```

- Currying this function enables the creation of new functions where one or more of the three unknowns is known, e.g.:

```
const curriedSubstr =
  from => numChars => str =>
    str.substr(from, numChars);
```

Array Methods

- JS arrays perfectly demonstrate the successful mixing of object-oriented and function programming techniques, e.g. the `push` method is stateful, the `concat` method if functional

- Array methods `map` and `filter` borrow from functional programming to facilitate the transformation of data

- The map method creates a new array by mapping each element of the existing one onto a new value, e.g.:
  ```
  nums.map(num => num += 1);
  ```

- The filter method creates a new array by filtering each element using a predicate, e.g.:
  ```
  nums.filter(num => num % 2 == 0);
  ```

Function Composition

- Function composition is the act of combining many functions into one, e.g.:

```
const toNumAndSquare =
  str => square(Number(str));
```

- It's possible to write a function that will compose functions, e.g.:

```
const compose = funcs => data =>
  funcs.reduce((val, func) => func(val), data);
```

- A pipeline is similar except that the functions are executed from right to left (see reduceRight)

Summary

- Functional programming favours functions, transformations, and immutable data
- Higher order functions are those that accept and/or return a function(s)
- Pure functions always produce the same output for the given input
- Immutability is hard to achieve in native JS but it is useful, in some circs, to treat your objects as if they were immutable
- Currying is the process of making new functions from an existing one
- Array methods map and filter, among others, are used to transform arrays (create new ones from existing ones)
- Function composition is the act of combining many functions into one