W07 - Further Functions

- Function Properties and Methods - functions are first-class objects & have properties and metho

.length → returns number of parameters.

call() and apply()

call() to set value of this inside a function

```
function sayHello() {
    return `Hello, my name is ${this.name}`;
}

const clark = { name: 'Clarke' }
const bruce = { name: 'Bruce' }

sayHello.call(clark); ← returns 'Hello, my name is Clarke'
sayHello.call(bruce); ← returns 'Hello, my name is Bruce'
```

- If the function requires parameters, they are included after the this parameter.
- If the function does not use this, call() can still be used but null is the first argument.

```
square.call(null, 4);
```

apply() is similar to call() but the arguments are provided in the form of an array. (useful if data is an arra

```
square.apply(null, [4]);
```

custom properties - can add properties to functions

```
square.description = 'Squares number provided as argument
```

Memoization - caching - allows quick retrieval if same argument is submitted a second time

```
function square(x) {
  square.cache = square.cache || {};
  if (!square.cache[x]) {
      square.cache[x] = x * x;
  }
  return square.cache[x];
}
```

WOT - Further Functions

- Immediately Invoked Expression (IIFE) - an anonymous function that is invoked as soon as it is defined.
  - achieved by placing parenthoses at end of function

```
(function () {
  const temp = 'World';
  console.log(`Hello ${temp}`);
})();
```

Keeps variable wrapped up within scope of the function.

Temporary variables - since there is no way to remove a variable from a scope once it has been declared, placing any code that uses temporary variables within an IIFE will ensure it is only available while the IIFE is invoked.

```
let a = 1;
let b = 2;
(() => {
  const temp = a;
  a = b;
  b = temp;
})();
```

a; ← 2

b; ← 1

console.log(temp); ← error: "temp not defined"

Initialization Code - IIFE can be used to set up initializatio

Safe Use of Strict Mode - Place all code in an IIFE with strict

```
(function () {
  'use strict';
    // all code goes here
})();
```

Ensures own code runs in strict mode without forcing other code into it.

Self contained code Blocks - IIFE can be used to enclose a block of code inside its own private scope.
  - discrete sectioning of parts of code

W07 - Further Functions

- Functions that Define and Rewrite Themselves

assign an anonymous function to a variable with the same name.

```
function party () {
  console.log ('Wow this is amazing!');
  party = function () {
    console.log ('Been there, got the T-Shirt');
  }
}
```

First time called, redefined to second console.log. Each consecutive call will just be second console.log

- If we create a variable, assigning the function to that variable before calling the function, the variable will retain the initial instance of the function.
- Properties set to this type of function before they are invoked, will be lost.

Init-Time Branching - can be used with feature detection on first run, then avoid checking on consecutive calls.

```
function ride() {
    if (window.unicorn) {
        ride = function () {
            // code that uses 'unicorn'
        }
    } else {
        ride = function () {
            // code that uses older technology
        }
    }
    return ride();
}
```

on first call, determines which function it should be, ends with calling the function to finish re-writing.

W07 - Further Functions

- Recursive Functions - invokes itself until a certain condition is met. Used for iterative processes

```
function factorial (n) {
  if (n===0) {
      return 1;
  }else{
    return n * factorial (n-1);
  }
}
```

If we know the loop may become eternal, we need to ensure our test accounts for this

- if argument is even, divide by two
- if argument is odd, multiply by Three and add One.

→ will end in 4,2,1, 4,2,1,...

```
function collatz (n, sequence=[n]) {
  if (n===1){
    return `Sequence took ${sequence.length} steps. It was
        ${sequence}`;
  }
  if (n%2===0) {
    n = n / 2;
  } else {
    n = 3 * n + 1;
  }
  return collatz (n, [...sequence, n]);
}
```

Starts by, first, checking whether n===1, returning a message with number of steps it took to get to 1. Otherwise, it checks for odd or even, invoking division or multiplication before calling itself again. the new sequence is constructed by placing the old sequence and the value of n inside a new array using the spread operator

W07 - Further Functions

- Callbacks - functions passed to other fuctions as arguments

Event-driven Asynchronous Programming - ensures waiting for an event to happen doesn't hold up execution of other parts of the program.

```
function wait (message, callback, seconds) {
    setTimeout (callback, seconds * 1000);
    console.log (message);
}

function selfDestruct () {
    console.log ('BOOOM!!');
}
```

```
wait ('This tape will self-destruct in five seconds...', selfDestruct, 5)
console.log ('Hmm, should I accept this mission?');
```

↳ "This tape will self-destruct in five seconds..."
↳ "Hmm, should I..."
↳ "BOOOM!"

Callback Hell - too much implementation of callbacks can result in "spaghetti programming" → large number of nested blocks

Promises - represents the future result of an asynchronous operation,
- to help simplify the process.

promise life cycle

called → pending (unsettled)
→ completed (settled)

settled promises have two possible outcomes: Resolved / Rejected

Super Promise

W07 - Further Functions

Creating a promise - use a constructor function with an executor as an argument, and two functions
resolve() called if operation is successful
reject() called if operation fails

```
const promise = new Promise( (resolve, reject) => {
    // initialization code here
    if (success) {
        resolve(value);
    } else {
        reject(value);
    }
});
```

Dice Example:

```
const dice = {  // object
    sides: 6,
    roll() {
        return Math.floor(this.sides * Math.random()) + 1;
    }
}

const promise = new Promise( (resolve, reject) => {
    const n = dice.roll();
    setTimeout(() => {
        (n > 1) ? resolve(n) : reject(n);
    }, n * 1000);
});
```

Dealing With a Settled Promise - then() method used to deal with the outcome. Two arguments - fulfilment function
- rejection function

```
promise.then( result => console.log(`Yes! I rolled a ${result}`),
              result => console.log(`Drat! ... I rolled a ${result}`) );
```

catch() method can be used to specify failure

```
promise.catch( result => console.log(`Drat! ... I rolled a ${result}`) );
```

chain then() and catch() to form succinct description

W07 - Further Functions

Chaining Promises - if multiple functions perform asynchronous operations returning promises, chain the then() methods forming sequencial code that is easy to read.

```
login(userName)
.then(user => getPlayerInfo(user.id))
.then(info => loadGame(info))
.catch(throw error)
```

Async Functions - allow to write asynchronous code as if it was syncronous.

async keyword using await operator before asynchronous fun

wrap the return value of the function in a promise that can be assigned to a variable. The next line of code is not executed until the promise is resolved.

```
async function loadGame(userName) {
    try {
      const user = await login(userName);
      const info = await getPlayerInfo(user.id);
      // load game using returned info
    }
    catch(error) {
      throw error;
    }
}
```

Generalized Functions - callbacks can be used to build more genealized functions.

W07 - Further Functions

- Functions returning Functions

In addition to being arguments, functions can also be returned.

```
function returnHello() {
    console.log('returnHello() called');
    return function(){
        console.log('Hello World!');
    }
}
```

When the function is called, logs the call then returns a function. this needs to be assigned to a variable.

```
const hello = returnHello(); ← logs 'returnHello() called'
```

invoke hello by placing parentheses after hello

```
hello(); ← logs 'Hello World!';
```

- Closures - powerful feature of JavaScript, if difficult to understa

Function Scope - variables are available within the function they are created

whenever a function is defined inside another function, the inner function has access to the outer function's variab

Generators - special functions used to produce iterators while maintaining the state of a value.

* placed after the function declaration

```
function* exampleGenerator(){
    // code for generator
}
```

calling this fuction doesn't actually run the code. It returns a Generator Object used to create an iterator to implement a next() method returning a value everytime next() is called

W07 - Further Functions

- Functional Programming

  functional languages include Clojure, Scala, Erlang

  Fundamental Elements of functional programming:

  - pass functions as arguments
  - return functions from other functions
  - use anonymous functions
  - closures

  object-Oriented, procedural, function programming are all paradigms of programming.

  JavaScript is a multi-paradigm language

  Java is all object-oriented.

Pure Functions - functions that adhere to these rules:

- return value only depends on the values provided as arguments. - DOES NOT RELY ON VALUES FROM SOMEPLACE ELSE IN THE PROGRAM -
- No side-effects. - DOES NOT CHANGE ANY VALUES OR DATA ELSEWHERE IN THE PROGRAM -
- Referential Transparency - GIVEN THE SAME ARGUMENT WILL ALWAYS RETURN SAME RESULT

A pure function must have:

- at least one argument - return value depends on the argument.
- a return value - otherwise, no point in the function.

Pure functions make functional programming more concise

W07 - Further Functions

Higher-Order Functions - functions that accept another function as an argument, or return another function as a result, or both.

- closures are important aspect of higher-order function
  - create generic functions that can be used to return more specific functions based on arguments

■ currying ← partial application of functions

- a function is curried when not all arguments are supplied to the function, so it returns a function that retains the arguments already provided, expecting the remaining arguments that were originally omitted
- relies on higher-order function capable of returning partially applied functions.

A General Curry Function - multiplier was hardcoded so it could be curried. can use curry() function to take any function and allow it to be partially applied.

GETTING FUNCTIONAL ← adopting some principles, such as keeping functions as pure as possible, keeping changes in state to a minimum will help improve standard of programming