Understand key JavaScript concepts

Aims

- Understand scope in JS (var, let, const)
- Functions
- Strings
- Arrays
- Describe the prototypical nature of all JavaScript-based inheritance
- Closure scope

Variable scope and hoisting in JS (var, let, const)

Problems with var:

- Scope
- Hoisting
- Redeclarion

`` `let` `` vs `` `const` ``

- Rule of thumb to prefer const over let
- const can't be reassigned
- Note the difference for Objects and Arrays

Functions

- Multi-paradigm nature of JS
- Functions passed as arguments
- Functions assigned as values in an object
- (Fat) Arrow Functions

Strings

- We can use different quotes to declare a string (single, double and backtick)
- Backtick defined strings can have interpolated values declared with `` `${}` ``.
- Strings are immutable but you can access characters
- Helper methods can transform strings and pass values back.

Arrays

- Define with `` `[]` ``.
- Key methods (.join(), .map(), .filter(), .reduce(), .forEach())
- Destructuring

Prototypical Inheritance

Inheritance with JS is achieved with a chain of prototypes. These approaches have evolved significantly over time.

The three common approaches to creating a prototypal chain:

- functional
- constructor functions
- class-syntax constructors

For the purposes of these examples, we will be using a Wolf and Dog taxonomy, where a Wolf is a prototype of a Dog.

Prototypical Inheritance (Functional)

```javascript
const wolf = {
  howl: function() { console.log(`${this.name} awoooooo`)}
}

const dog = Object.create(wolf, {
  woof: {value: function() {console.log(`${this.name} woof`)}}
})

const rufus = Object.create(dog, {
  name: {value: 'Rufus the dog'}
})

rufus.woof()
rufus.howl()
```

Prototypical Inheritance (Constructor function)

```javascript
function Wolf(name) {
  this.name = name;
}

Wolf.prototype.howl = function() {
  console.log(`${this.name} awoooooooo`)
}

function Dog(name) {
  Wolf.call(this, `${name} the dog`)
}

Object.setPrototypeOf(Dog.prototype, Wolf.prototype)

Dog.prototype.woof = function() {
  console.log(`${this.name} woof`)
}

const rufus = new Dog('Rufus')

rufus.woof()
rufus.howl()
```

Prototypal Inheritance (Class-Syntax Constructors)

```javascript
class Wolf {
  constructor(name) {
    this.name = name
  }
  howl() {
    console.log(`${this.name} awoooooooo`)
  }
}

class Dog extends Wolf {
  constructor(name) {
    super(`${name} the dog`)
  }
  woof() {
    console.log(`${this.name} woof`)
  }
}

const rufus = new Dog('Rufus')

rufus.woof()
rufus.howl()
```

Closure Scope (1/3)

When a function is created, an invisible object is also created - this is the closure scope.

Parameters and variables created in the function are stored on this object.

```
function outerFunction() {
  const foo = true;
  function print() {
    console.log(foo)
  }
  foo = false
  print()
}
outerFunction()
```

Closure (2/3)

If there is naming collision then the reference to nearest close scope takes precedence.

```
function outerFn () {
  var foo = true
  function print(foo) {
    console.log(foo)
  }
  print(1) // prints 1
  foo = false
  print(2) // prints 2
}
outerFn()
```

In this case the foo parameter of print overrides the foo var in the outerFn function.

Closure Scope (3/3)

Closure scope cannot be accessed outside of a function.

```
function outerFn () {
  var foo = true
}
outerFn()
console.log(foo) // will throw a ReferenceError
```

Since the invisible closure scope object cannot be accessed outside of a function, if a function returns a function the returned function can provide controlled access to the parent closure scope.

```
function init (type) {
  var id = 0
  return (name) => {
    id += 1
    return { id: id, type: type, name: name }
  }
}
```

Exercises

There are a number of exercises for you to work on. These are all found in `` `Labs/Student/02-key-js-concepts` ``. There are corresponding solutions in `` `Labs/Solutions/02-key-js-concepts` ``.

Each of them have tests, so to check you’ve got it right run `` `node filename` `` in your terminal.

Creating an Event Emitter

The events module exports an EventEmitter constructor:

```
const {EventEmitter} = require('events')
```

and, now the `events` module is the constructor as well:

```
const EventEmitter = require('events')
```

So to create a new event emitter:

```
const myEmitter = new EventEmitter()
```

A more typical pattern is to inherit from the EventEmitter.

```
class MyEmitter extends EventEmitter {
  constructor (opts = {}) {
    super(opts)
    this.name = opts.name
  }
}
```

Emitting Events

```
const { EventEmitter } = require('events')
const myEmitter = new EventEmitter()
myEmitter.emit('an-event', some, args)
```

An example of using emit with inheriting from EventEmitter:

```javascript
const { EventEmitter } = require('events')
class MyEmitter extends EventEmitter {
  constructor (opts = {}) {
    super(opts)
    this.name = opts.name
  },
  destroy (err) {
    if (err) { this.emit('error', err) }
    this.emit('close')
  }
}
```

Listening for Events

To add a listener, use the addListener method.

```
const { EventEmitter } = require('events')

const ee = new EventEmitter()
ee.on('close', () ⇒ { console.log('close event fired!') })
ee.emit('close')
```

It could also be written as:

```
ee.addListener('close', () ⇒ {
  console.log(close event fired!')
})
```

Arguments passed to emit are received by the listener function.

```
ee.on('add', (a, b) ⇒ { console.log(a + b) }) // logs 13
ee.emit('add', 7, 6)
```

Order is important

This listener will not fire:

```
ee.emit('close')
ee.on('close', () ⇒ { console.log('close event fired!') })
```

Listeners are called in the order they are registered:

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()
ee.on('my-event', () ⇒ { console.log('1st') })
ee.on('my-event', () ⇒ { console.log('2nd') })
ee.emit('my-event')
```

But the `prependListener` method can be used to inject listeners to the top position:

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()
ee.on('my-event', () ⇒ { console.log('2nd') })
ee.prependListener('my-event', () ⇒ { console.log('1st') })
ee.emit('my-event')
```

Single or Multi-use

An event can be used more than once:

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()
ee.on('my-event', () => { console.log('my-event fired') })
ee.emit('my-event')
ee.emit('my-event')
ee.emit('my-event')
```

The once method will immediately remove its listener after it has been called.

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()
ee.once('my-event', () => { console.log('my-event fired') })
ee.emit('my-event')
ee.emit('my-event')
ee.emit('my-event')
```

Removing Listeners

The removeListener method can be used to remove a previously registered listener.

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()

const listener1 = () ⇒ { console.log('listener 1') }
const listener2 = () ⇒ { console.log('listener 2') }

ee.on('my-event', listener1)
ee.on('my-event', listener2)

setInterval(() ⇒ {
  ee.emit('my-event')
}, 200)

setTimeout(() ⇒ {
  ee.removeListener('my-event', listener1)
}, 500)

setTimeout(() ⇒ {
  ee.removeListener('my-event', listener2)
}, 1100)
```

Remove all listeners

The removeAllListeners method can be used to remove listeners without having a reference to the function.

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()

const listener1 = () ⇒ { console.log('listener 1') }
const listener2 = () ⇒ { console.log('listener 2') }

ee.on('my-event', listener1)
ee.on('my-event', listener2)
ee.on('another-event', () ⇒ { console.log('another event') })

setInterval(() ⇒ {
  ee.emit('my-event')
  ee.emit('another-event')
}, 200)

setTimeout(() ⇒ {
  ee.removeAllListeners('my-event')
}, 500)

setTimeout(() ⇒ {
  ee.removeAllListeners()
}, 1100)
```

The Error Event

What will happen here?

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()

process.stdin.resume() // keep process alive

ee.emit('error', new Error('oh oh'))
```

Emitting an 'error' event on an event emitter will cause the event emitter to throw an exception if a listener for the 'error' event has not been registered.

```
const { EventEmitter } = require('events')
const ee = new EventEmitter()

process.stdin.resume() // keep process alive

ee.on('error', (err) => {
  console.log('got error:', err.message )
})

ee.emit('error', new Error('oh oh'))
```