JS The Tricky Parts

Goals

- Introduce and review some of the more complex concepts in JS
- Make sure you're ready for interview questions!

JS The Must-Know Parts

- Array / String / Object / Number methods
- type checking / conversion
- JS operators (`==` vs `===`) etc.

Asynchronous Code

- AJAX
- XHR / Fetch
- Callbacks / Promises / Async Await

JS The Tricky Parts

Closures

- The ability for inner functions to remember variables defined in outer functions, long after the outer function has returned
- Useful for encapsulating logic and creating private variables

An Example

```
function idGenerator() { let start = 0; return function generate() { start++;
return start; }; }
```

- Can you spot the closure?

IIFE

Immediately Invoked Function Expression

```
(function() { console.log('just ran!'); })();
```

- Useful for scoping something right away and protecting the global namespace

IIFE + Closure

```
const $ = (function() { const version = '3.1.4'; return { displayVersion() {
return version; }, html(elem) { return
document.querySelector(elem).innerHTML; } }; })();
```

- Can you spot the closure?

JS OO Under the Hood

- `new`
- prototypes
- constructor
- `Object.getPrototypeOf`

- prototypal inheritance

`new`

- The keyword does four things:

  new

  1. Creates an empty object
  2. Sets the keyword ***this*** to be that object
  3. Returns the object -***return this***
  4. Creates a link to the object's prototype

Creates a Link?

Before we get there - let's review objects/functions in JS

- Every **function** has a property on it called prototype
- The prototype object has a property called constructor which points back to the function
- When the ***new*** keyword is used to invoke a function, a link between the object created from new and the prototype object is established
- This link is called the **internal prototype** and can be accessed using `Object.getPrototypeOf()`

💡 **Note:** Previously, people used to get the prototype by accessing a property called __proto__. Based on the name of this property alone, you can probably guess it was never intended for direct access and use, but you know, JavaScript programmers. Anyway, this way of getting a prototype is *officially deprecated*.

Show Me Some Code!

```
function Vehicle(make, model, year) { this.make = make; this.model = model;
this.year = year; } Vehicle.prototype; // an object
Vehicle.prototype.constructor === Vehicle; // true let myFirstCar = new
Vehicle('Toyota', 'Corolla', 2005); Object.getPrototypeOf(myFirstCar) ===
Vehicle.prototype; // true
```

Functions on the Prototype

It's better to create instance methods on the prototype instead of defining them in the constructor.

- Why?

Consider the Following

```
// ES5 function Vehicle(make, model, year) { this.make = make; this.model =
model; this.year = year; this.start = function() { return 'Starting!'; }; }
Vehicle.prototype.honk = function() { return 'Beep!'; };
```

```
// ES2015 class Vehicle { constructor(make, model, year) { this.make = make;
this.model = model; this.year = year; this.start = function() { return
'Starting!'; }; } honk() { return 'Beep!'; } }
```

The Purpose of the Prototype

- JavaScript uses this object to find methods and properties on everything in JS!
- If a property can not be found, JS works it's way up the "prototype chain", finding the prototype of every object
- If the property can not be found, undefined is returned

Prototypal Inheritance

```
function Vehicle(make, model, year) { this.make = make; this.model = model;
this.year = year; } Vehicle.prototype.honk = function() { return 'Beep!'; };
function Car(make, model, year) { Vehicle.call(this, make, model, year); //
similar to "super(make, model, year)" } Car.prototype =
Object.create(Vehicle.prototype); Car.prototype.constructor = Car;
```

Notes on ES5 OOP

- ES2015 does all of this under the hood
- Make sure you're able to explain what a prototype is
- Be able to define the prototype chain, how inheritance can be implemented

Semicolons!

- Trivia time!
- If you don't add a semi-colon, JS will automatically insert one (also known as ASI)

Code

```
function createPerson(first){ return {first} } function
createPersonNewLine(first){ return {first} } createPerson('Steph') // {first:
'Steph'} createPersonNewLine('Steph') // undefined
```

Functional Programming

- FP is the process of building software by composing pure functions, avoiding: shared state, mutable data, and side-effects.
- FP is often declarative rather than imperative, and application state flows through pure functions.

OOP vs FP

- OOP is typically easier to reason about and read.
- FP has a much steeper learning curve, but can allow for functions to be simplified and easily composed.

Essential Concepts

- pure functions
- closure
- function composition
- partial application / currying
- HOFs, First-Class Functions

Design Patterns

Agreed upon standards / best-practices

- module pattern
- singleton pattern
- many others!

JS The Trivia Parts

- var / let / const
- new keyword
- keyword this
- reference types
- immutability
- hoisting
- what does this output?
- call / apply / bind
- arrow functions / bind
- setTimeout 0
- for loop with closure

Loops with closure

```
for(var i = 0; i < 5; i++){ setTimeout(function(){ console.log(i) }, 1000) }
```

Issues here

- *i* is scoped globally
- by the time the setTimeout runs, the value is 5
- We can fix this using the ***let*** keyword or writing an IIFE

Where you can learn more / practice

- Advanced Web Developer Udemy course (sections on closure + OOP)
- Rithm School curriculum
- Anything by Eric Elliot + Brian Lonsdorf for functional programming
- https://30secondsofinterviews.org/
- https://drboolean.gitbooks.io/mostly-adequate-guide-old/content/
- https://blog.bitsrc.io/understanding-design-patterns-in-javascript-13345223f2dd