WEEK 3 JavasScript:

JS:

Exercise 3.1:
Create a memoize function that remembers previous inputs and stores them in cache so that it won't have to compute the same inputs more than once. The function will take an unspecified number of integer inputs and a reducer method. **(1.5 hours)**

Example:

```
//Given reducer method:
function add(a,b){
 return a+b
}
//Create a method called memoize such that:
const memoizeAdd = memoize(add);
//then calling…
memoizeAdd(100,100); //returns 200
memoizeAdd(100); //returns 100
memoizeAdd(100,200) //returns 300
memoizeAdd(100,100) //returns 200 without computing
```

Guidelines:

1. The memozie function should be written from scratch.
2. 3rd party libraries such as loaddash or underscore should not have been used.
3. The function should carry a name which should denote the functionality of it.
4. The function should be able to take n number of arguments.

Outcomes:

1. Understanding how caching works.
2. Why is it important?
3. The problem memoization solves.

Exercise 3.2:
Create 3 simple functions where call, bind and apply are used. The intention of this exercise is to understand how they work and their differences. **(0.5 hours)**

Guidelines:

1. The candidate should be able to explain what call, bind and apply in JS are and its differences.
2. Using live examples the candidate should be able to differentiate between them.

Outcome:

1. Call, Bind, Apply are very important JS concepts.
2. This exercise should help understand them and also understand the differences between them.
3. Writing examples for the same will help the candidate have a better understanding of when and how they are used in real world scenarios.

Exercise 3.3:

1. What is the output of the below problem and why**:[30 min]**

```
function createIncrement() {
  let count = 0;
  function increment() {
    count++;
  }
  let message = `Count is ${count}`;
  function log() {
    console.log(message);
  }

  return [increment, log];
}
const [increment, log] = createIncrement();
increment();
increment();
increment();
log(); // What is logged?
```

Guidelines:

1. The candidate should be able to explain the code with the desired output.

Outcome:

1. The candidates will understand how ‘closure’ works in JS.
2. The candidates will understand how ‘encapsulation’ works in JS.

Exercise 3.4

Refactor the above stack implementation, using the concept of closure, such that there is no way to access items array outside of createStack() function scope: (2-3 hours)

```
function createStack() {
  return {
    items: [],
    push(item) {
      this.items.push(item);
    },
    pop() {
      return this.items.pop();
    }
  };
}
const stack = createStack();
stack.push(10);
stack.push(5);
stack.pop(); // => 5
stack.items; // => [10]
stack.items = [10, 100, 1000]; // Encapsulation broken!
```

```
function createStack() {
  // Write your code here...
}
const stack = createStack();
stack.push(10);
stack.push(5);
stack.pop(); // => 5
stack.items; // => undefined
```

Guidelines:

2. The candidate should be able to refactor the code and get the desired output.
3. The candidate should be able to explain the code why it was not working before.
4. The candidate should be able to explain the code why it is working now.

Outcome:

3. The candidates will understand how ‘closure’ works in JS.
4. The candidates will understand how ‘encapsulation’ works in JS.