JS Solid! - 2

Call stack
CallBack Queues
Event Loop
Closures,
Factory Functions

Objective

1. explain common Algorithms and Data Structures
2. understanding how js works under the hood
3. understanding closures and how to use them
4. Performance optimization
5. stacks and queues

reasoning behind good code

- 1. Performance : speed of execution
- 2. Readability : ease of understanding
- 3. Maintainability : ease of updating
- 4. Scalability : ability to handle large amounts of data
- 5. Reusability : ability to be used in other projects

call stack and call back queue

- call stack is a data structure that uses the last in first out (LIFO) principle to temporarily store and manage function invocation (call).
- call back queue is a data structure that uses the first in first out (FIFO) principle to temporarily store and manage function invocation (call).

example of call stack and call back queue

```javascript
// function for call stack
function first() {
  second();
  console.log('Hi there');
}
// function for call back queue
function second() {
  setTimeout(() => {
    console.log('Async');
  }, 2000);
  console.log('The end');
}
first();
// output
// The end
// Hi there
// Async
```

Explanation for the example

- `first()` is called and pushed to the call stack
- `second()` is called and pushed to the call stack
- `setTimeout()` is called and pushed to the call back queue
- `console.log('The end')` is called and executed?
- `second()` is popped off the call stack
- `console.log('Hi there')` is called and executed
- `first()` is popped off the call stack
- `setTimeout()` is popped off the call back queue and pushed to the call stack
- `console.log('Async')` is called and executed

memory heap

- memory heap is a place where the memory allocation happens in javascript
- memory allocation is the process of assigning memory to a variable or function
- memory allocation is done by the garbage collector
- garbage collector is a process that frees up memory when it is not being used

```
const a = 1;
// in memory heap a = 1 is stored in memory and a is pointing to that memory location
```

JS execution

- JS is a single threaded language
- the current task being executed is called the the thread of execution which continues line by line

Execution example

- `const number = 1;` is executed and stored in memory heap
- `function multiplyBy2(inputNumber) {` is executed and stored in memory heap
- `const result = inputNumber * 2;` is executed and stored in memory heap
- `return result;` is executed and stored in memory heap
- `const name = 'Will';` is executed and stored in memory heap
- `multiplyBy2(4);` is executed and stored in memory heap
- `multiplyBy2(4);` is popped off the call stack

```
const number = 1;
function multiplyBy2(inputNumber) {
  const result = inputNumber * 2;
  return result;
}
const name = 'Will';
multiplyBy2(4);
```

global execution context

- global execution context is the default context in which code is executed in javascript
- global execution context is created in two phases
- 1. creation phase
    - creation phase is where the memory is allocated for variables and functions
- 2. execution phase
    - execution phase is where the code is executed line by line

creation phase

- creation phase is where the memory is allocated for variables and functions
- creation phase is done in two steps
- 1. creation of the global object/global variable
    - global object is the window object in the browser
    - global object is the global object in node
    - var a = 1; is stored in the global object as a property
- 2. creation of the this keyword
    - this keyword is a reference to the global object
    - this keyword is used to access the global object

execution phase

- execution phase is where the code is executed line by line
- execution phase is done in two steps

1. hoisting

- hoisting is the process of moving all the variable and function declarations to the top of their scope before code execution
- hoisting is done for both function declarations and variable declarations
- hoisting is done for variable declarations but not for variable assignments
- hoisting is done for function declarations but not for function assignments

hoisting example

```
function wlecomeStudent() {
  console.log('Welcome to the class');
  function rollCall() {
    console.log('Roll call');
    return;
  }
  console.log('Begin Roll call');
  rollCall();
}
console.log('Start class');
welcomeStudent();
console.log('End class');
// output
// Start class
// Welcome to the class
// Begin Roll call
// Roll call
// End class
```

Hoisting example walk-through

- `function welcomeStudent() {` is hoisted to the top of the scope and stored in memory heap
- TOE thread of execution than executes the console log `console.log('Start class');` because it is was invoked
- `function welcomeStudent()` is invoked and pushed to the call stack
- TOE now enters the function welcomeStudent() and executes the console log `console.log('Welcome to the class');`
- TOE now executes the function `function rollCall()` and pushes it to the call stack
- one the function is invoked a new local execution context is created
- TOE now executes the console log `console.log('Roll call');` and pops off the function rollCall() from the call stack
- finally the output looks like

```
Start class
Welcome to the class
Begin Roll call
Roll call
End class
```

console.trace()

- console.trace() is a method that prints the stack trace of the current point in the code
- stack trace is a list of all the functions that are currently on the call stack

stack trace example

```
function first() {
  second();
  console.log('Hi there');
}
function second() {
  third();
  console.log('The end');
}
function third() {
  console.trace();
  console.log('Trace');
}
first();
```

stack trace example walk-through

- `function first()` is hoisted to the top of the scope and stored in memory heap
- `function second()` is hoisted to the top of the scope and stored in memory heap
- `function third()` is hoisted to the top of the scope and stored in memory heap
- TOE executes the function `first()` and pushes it to the call stack
- TOE executes the function `second()` and pushes it to the call stack
- TOE executes the function `third()` and pushes it to the call stack
- TOE executes the console.log `console.log('Trace');` and pops off the function `third()` from the call stack
  - console.trace will display the stack trace of the current point in the code
  - output

```
at third (REPL38:2:11)
at second (REPL33:2:3)
at first (REPL29:2:3)
```

Asynchronous nature of javascript

- web API
- Event Loops
- call back queue
  - microtask queue
  - timer queue
  - I/O queue
  - animation queue
  - network queue

Call back que priority

- top priority is the micro task queue: these are tasks that are executed immediately after the current task is completed, example `Promise`, `MutationObserver`, `process.nextTick`
- timer queue: these are tasks that are executed after a specified amount of time, example `setTimeout`, `setInterval`
- I/O queue: these are tasks that are executed after a specified amount of time, example `XMLHttpRequest`, `fetch`
- nimation queue: these are tasks that are executed after a specified amount of time, example `requestAnimationFrame`
- network queue: these are tasks that are executed after a specified amount of time, example `WebSocket`, `EventSource`
- idle queue: these are tasks that are executed after a specified amount of time, example `requestIdleCallback`
- check queue: these are tasks that are executed after a specified amount of time, example `setImmediate`
- finally the micro task queue is checked again and the process repeats
- The ques are executed on FIFO basis, first in first out

call back queue example

```
console.log('Start');
setTimeout(() => {
  console.log('Set timeout');
}, 0);
Promise.resolve().then(() => {
  console.log('Promise');
});
console.log('End');
// output
// Start
// End
// Promise
// Set timeout
```

call back queue example walk-through

- `console.log('Start');` is executed and output is `Start`
- `setTimeout(() => { console.log('Set timeout'); }, 0);` is executed and pushed to the timer queue
- `Promise.resolve().then(() => { console.log('Promise'); });` is executed and pushed to the micro task queue
- `console.log('End');` is executed and output is `End`
- The micro task queue is checked and the function `() => { console.log('Promise'); }` is executed and output is `Promise`, the reason why promise is executed before the timer queue is because the micro task queue has a higher priority than the timer queue.
- The timer queue is checked and the function `() => { console.log('Set timeout'); }` is executed and output is `Set timeout`
- finally the output looks like

```
Start
End
Promise
Set timeout
```

Closures, Factory Functions

Declarative vs imperative programming

- declarative programming is a style of programming where you tell the computer what you want to do and not how to do it. i.e What a program should do rather than how it should do it.
- imperative programming is a style of programming where you tell the computer how to do something. i.e How a program should do it rather than what it should do it.
  Example of imperative programming is js

first order Objects

- first order objects are objects that can be assigned to a variable, passed as an argument to a function, returned from a function and have methods.
- Function aer first order objects in js
    - store value
    - pass arguments
    - return value

Closures

- Closures are functions that have access to the parent scope even after the parent function has closed.
- When a function is executed an execution context is created, when a function is returned a closure is created.

Closure example

```
function outer() {
  let counter = 0;
  function incrementCounter() {
    counter++;
    console.log(counter);
  }
  return incrementCounter;
}
const myfunc = outer();
myfunc(); // 1
myfunc(); // 2
// output
// 1
// 2
```

Closure example walk-through

- `function outer()` is hoisted to the top of the scope and stored in memory heap
- TOE executes the function `outer()` and pushes it to the call stack
- TOE executes the function `incrementCounter()` and pushes it to the call stack
- TOE executes the console.log `console.log(counter);` and pops off the function `incrementCounter()` from the call stack
- TOE executes the return `return incrementCounter;` and pops off the function `outer()` from the call stack
- `const myfunc = outer();` is executed and the function `incrementCounter()` is returned and assigned to the variable `myfunc`
- `myfunc();` is executed and the function `incrementCounter()` is executed and the counter is incremented and output is `1`
- `myfunc();` is executed and the function `incrementCounter()` is executed and the counter is incremented and output is `2`
- finally the output looks like

```
1
2
```

lexical scope and lexical environment

- lexical scope is determined when the during runtime when hs is complied.
- lexical env is the environment where the ref to the variable is stored during program execution

closure example 2

```
function privateCounter() {
  let count = 0;
  function changeBy(val) {
    count += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return count;
    }
  };
}
const counter1 = privateCounter();
const counter2 = privateCounter();
console.log(counter1.value()); // 0
counter1.increment();
counter1.increment();
console.log(counter1.value()); // 2
counter1.decrement();
console.log(counter1.value()); // 1
console.log(counter2.value()); // 0
// output
```

Factory functions

- Factory functions are functions that return objects, they are used to create multiple objects with the same properties and methods.
- Instead of returning functions, however, we are returning object literals.

Factory function example

```
function createCircle(radius) {
  return {
    radius,
    draw: function() {
      console.log('drawing', radius *= 2);
    }
  };
}
const circle1 = createCircle(10);
const circle2 = createCircle(20);
console.log(circle1);
circle1.draw();
console.log(circle2);
circle1.draw();
//********** output **********
// { radius: 10, draw: [Function: draw] }
// drawing 20
// { radius: 20, draw: [Function: draw] }
// drawing 40
```

Factory function example walk-through

- the function createCircle is passed the argument `10` and the function is executed and the object literal is returned and assigned to the variable `circle1`