JavaScript Submission 3

1. What is “closure” in JavaScript? Can you provide an example?

   In JavaScript, closure is a feature that allows a function to access variables from an outer function even after the outer function has finished executing. This is possible because the inner function ‘closes over’ the variables it needs from the outer function, even if the outer function’s execution context has been destroyed.
   Closure is the powerful concept in JavaScript that allows for the creation of private variables and methods, data encapsulation, and the implementation of design patterns like the module pattern.

   Example:

```
function outerFunction(x){
    function innerFunction(y){
        return x+y;
    }
    return innerFunction;
}
const add5 = outerFunction(5);
console.log(add5(3));    //output: 8
```

   In this example, The ‘outerFunction’ takes a parameter ‘x’ and returns an inner Function ‘innerFunction’. Then the innerFunction takes a parameter ‘y’ and returns x + y. when ‘outerFunction(5)’ is called, it returns the ‘innerFunction’ and assigns it to the variable ‘add5’. Even though the outerFunction has finished executing and its execution context has been destroyed, the innerFunction still has access to the variable ‘x’ from the outer function due to closure.
   When add5(5) is called, the innerFunction can access the variable ‘x’ with the value ‘5’ that was passed to the outerFunction earlier, and it returns 8.

2. What are promises and how are they useful?

   Promises in JavaScript are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a cleaner and more manageable way to handle asynchronous code compared to traditional callback functions.
   Key features of promises:
   - A promise can be in one of three states : pending , fulfilled, rejected.
   - When a promise is fulfilled, it returns a value that can be used for further processing .
   - If a promise is rejected, it indicates an error occurred and provides a reason for the failure.
   - Promises allow you to chain multiple asynchronous operations together using `.then()` and `.catch()` methods.

- They help avoid the "callback hell" problem that occurs with deeply nested callbacks

Promises are useful for:

- Managing asynchronous operations like API calls, file I/O, or timeouts
- Chaining multiple asynchronous operations together
- Handling errors in asynchronous code in a more structured way
- Improving code readability and maintainability compared to nested callbacks

To create a promise, use the 'promise' constructor and pass in a function with 'resolve' and 'reject' parameters. Inside the function, you perform the asynchronous operation and call 'resolve' with the result or 'reject' with an error. Promises are widely used in modern Java Script development and are a core part of the language's asynchronous programming model.

3. How to check whether a key exists in a JavaScript object or not.

   There are several ways to check if a key exists in a JavaScript object.

   - Using the in operator:

     This operator returns 'true' if the specified property is in the object. It returns 'false' if the property is not in object.

```
const obj = { name: 'John', age: 30 };
console.log('name' in obj); // true
console.log('address' in obj); // false
```

   - Using the hasOwnProperty() method :

     This method returns 'true' if the object has the specified property as a direct property of that object and not inherited through the prototype chain.

```
const obj = { name: 'John', age: 30 };
console.log(obj.hasOwnProperty('name')); // true
console.log(obj.hasOwnProperty('address')); // false
```

   - Using the Object.keys() method :

     This method returns an array of the object's own enumerable string-keyed property names. Can then check if the desired key is included in this array.

```
const obj = { name: 'John', age: 30 };
console.log(Object.keys(obj))// ['name', 'age']
console.log(Object.keys(obj).includes('name')); // true
console.log(Object.keys(obj).includes('address')); // false
```

   - Using the typeof operator :

     This approach checks if the value associated with the key is not 'undefined'. However, it's not a foolproof method as the value could be set to 'undefined' intentionally.

```
const obj = { name: 'John', age: 30 };
console.log(typeof obj.name !== 'undefined'); // true
console.log(typeof obj.address !== 'undefined'); // false
```

4. What is the output of this code? Please explain.

```
var employeeId = 'abc123';

function foo() {
employeeId();
return;

function employeeId() {
console.log(typeof employeeId);
}
}
foo();
```

Output : Function

Explanation : In JavaScript , function declarations are hoisted to the top of their scope, overriding any variables with the same name. inside the foo() function , EmployeeId() refers to the inner function, not the outer variables. When typeof employeeId is logged, it outputs ‘function’, since the local function takes precedence over the outer variable.

5. What is the output of the following? Please explain.

```
(function() {
    'use strict';

    var person = {
    name: 'John'
    };
    person.salary = '10000$';
    person['country'] = 'USA';

    Object.defineProperty(person, 'phoneNo', {
    value: '8888888888',
    enumerable: true
    })

    console.log(Object.keys(person));
    })();
```

Output : ['name', 'salary', 'country', 'phoneNo']

Explanation : The function is an Immediately Invoked Function Expression -IIFE, which runs immediately after it’s defined. Object.keys(person) returns an array of all enumerable properties of the person object. Here, The user strict directive enforces stricter parsing and error handling and Object.defineProperty is used to add phoneNo as an enumerable property.

6. What is the output of the following? Please explain.

```
(function() {
    var objA = {
    foo: 'foo',
    bar: 'bar'
    };
    var objB = {
    foo: 'foo',
    bar: 'bar'
    };
    console.log(objA == objB);
    console.log(objA === objB);
    }());
```

Output :

```
false
false
```

Explanation :

console.log(objA == objB) - This compares objA and objB using the == operator, which checks for equality in terms of reference. Since objA and objB are two distinct objects (even though their contents are identical), they occupy different memory locations. Therefore, objA == objB evaluates to False.

console.log(objA === objB) - This uses the === operator, which checks for strict equality. Strict equality in JavaScript compares both the values and the types of the operands. Since objA and objB are two distinct objects with the same contents, they are not strictly equal. Hence , objA === objB also evaluates to false.

7. What is the output of the following? Please explain.

```
function Person(name, age){
    this.name = name || "John";
    this.age = age || 24;
    this.displayName = function(){
    console.log(this.name);
    }
    }

    Person.name = "John";
    Person.displayName = function(){
    console.log(this.name);
    }
    var person1 = new Person('John');
    person1.displayName();
    Person.displayName();
```

Output :   John
           Person

8. In-Class Exercise: Designing a School Management System
   Scenario:
   You are tasked with designing a School Management System for a school. The system should
   manage students, teachers, courses, and their interactions.
   Exercise Instructions:
   1. Identify Classes:
      - List down the main entities (classes) that you think are necessary for the School Management System. Consider entities like Student, Teacher, Course, etc.

   2. Define Class Properties:
      - For each identified class, define the properties (attributes) that would be essential to store information. For example, Student class might have properties like id, name, email, etc.

   3. Define Class Methods:
      - Specify the methods (functions) that each class should have. Think about what actions each class needs to perform. For instance, Student might need methods like enroll(course), getGrades(), etc.

   4. Class Relationships:
      - Determine how classes will interact with each other. For example, how will a Teacher assign a Course to a Student? How will a Course keep track of enrolled Students?

   5. Write Sample Code:
      - Write a basic implementation in JavaScript using classes and methods you've defined. This step can help reinforce understanding through practical application.

```
class Student {

    constructor(id,name,address,email){
        this.id = id;
        this.name = name;
        this.address = address;
        this.email = email;
        this.courses = [];
    }
```

```javascript
    enroll(courseId){
        this.courses.push(courseId);
    }
    getGrade(){
        return grades.filter(grade=>grade.StudentId === this.id);
    }
}

class Teacher {
    constructor(id,name,email){
        this.id = id;
        this.name = name;
        this.email = email;
        this.coursesTaught = [];
    }
    assignCourse(courseId,StudentId){

    }
    gradeStudent(StudentId,courseId,grade){
        grades.push({StudentId,courseId,grade})
    }
}

class Course {
    constructor(id,name,teacher){
        this.id = id;
        this.name = name;
        this.teacher = teacher;
        this.studentsEnrolled = [];
    }
    addStudent(StudentId){
        this.studentsEnrolled.push(StudentId);
    }
    listStudents(){
        return this.studentsEnrolled;
    }
}

let student1 = new Student(1,'Harish','colombo','harish@example.com');
let teacher1 = new Teacher(101,'Rihana','rihana2@example.com');
let course1 = new Course(1001,'JavaScript',teacher1);

student1.enroll(course1.id);
course1.addStudent(student1.id);

console.log(student1);  // Output : Student {id: 1, name: 'Harish',
address: 'colombo', email: 'harish@example.com', courses: Array(1)}

console.log(course1.listStudents()); //Output : [1]
```