JavaScript Submission 3

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

    - Closure in JavaScript is a form of lexical scoping used to preserve variables from the outer scope of a function in the inner scope of a function.

```
function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
        console.log('Outer variable: ' + outerVariable);
        console.log('Inner variable: ' + innerVariable);
    }
}

const newFunction = outerFunction('outside');
newFunction('inside');
```

2. What are promises and how are they useful?

    - Promises in JavaScript are a way to handle asynchronous operations, like fetching data from a server, in a cleaner and more manageable way.

    - Promises help to write code that runs in the background (asynchronously) without blocking the rest of the code. They also make it easier to handle errors and chain multiple asynchronous operations together.

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

    - Using the ‘**in**’ Operator

```
let obj = { key1: "value1", key2: "value2" };

console.log("key1" in obj); // true
console.log("key3" in obj); // false
```

- Using '**hasOwnPropertyt'** Method

```
let obj = { key1: "value1", key2: "value2" };

console.log(obj.hasOwnProperty("key1")); // true
console.log(obj.hasOwnProperty("key3")); // 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

Inside the **foo** function, the **employeeId** function is hoisted to the top. This means the employeeId function is known and available throughout the entire foo function, even before its actual line of definition.

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”]

An IIFE (Immediately Invoked Function Expression) with **use strict** creates an object **person** with a `name` property set to **John**. It then adds **salary** ('10000$') and **country**` ('USA') properties to the object. Using **Object.defineProperty**, it adds a **phoneNo** property ('8888888888') and makes sure it can be listed. The **Object.keys(person)** method returns an array of all the listable properties of the **person** object, which are **["name", "salary", "country", "phoneNo"].**

6. What is the output of the code? 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

objA and **objB** have identical properties. Both **`==`** and **`===`** comparisons between objA and **objB** evaluate to **false** because they reference different objects.

`objA == objB` is **false** because **objA** and **objB** are different objects.

`objA === objB` is also **false** because strict equality checks for reference equality, and **objA** and **objB** are not the same reference.

7. What is the output of the following code:

```
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

: john

The `**Person**` constructor function creates objects with optional `**name**` and `**age**` parameters, defaulting to "**John**" and 24 respectively. It also defines an instance method `**displayName**` that logs the object's `**name**`. Additionally, static properties directly attached to the `**Person**` function include a `**name**` property and a method to log it, which can be accessed without needing to create an instance of `**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.