Assignment – 1

- 26/ July/ 2024, Friday

1) What is a closure in JavaScript?

Ans: -

In JavaScript, a closure is a feature that allows a function to "remember" the environment in which it was created. In simple terms, it's when a function keeps access to variables from its outer scope, even after the outer function has finished executing.

Simple Explanation

Imagine you have a function inside another function. The inner function can access variables from the outer function. Even after the outer function has completed, the inner function still has access to those variables. This is because the inner function "closes over" the variables from the outer function.

Example

Let's break it down with a simple example:

```javascript
function createCounter() {
  let count = 0; // This is a variable in the outer function's scope

  return function() { // This is the inner function
    count += 1; // The inner function can access and modify 'count'
    return count; // Returns the updated count
  };
}

const counter = createCounter(); // 'counter' is now a function
console.log(counter()); // Outputs: 1
console.log(counter()); // Outputs: 2
console.log(counter()); // Outputs: 3
```

How It Works

1. `` `createCounter()` `` **Function**: When you call `` `createCounter()` ``, it creates a local variable `` `count` `` and then returns an inner function.
2. **Returning the Inner Function**: The inner function has access to `` `count` `` because it was created inside `` `createCounter()` ``, so it forms a closure.
3. **Using the Closure**: When you call the `` `counter()` `` function (which is actually the inner function returned by `` `createCounter()` ``), it can still access and modify the `` `count` `` variable, even though `` `createCounter()` `` has finished running.

In summary, closures let functions keep track of variables from their parent functions, allowing for powerful patterns in JavaScript like data encapsulation and function factories.

---

2) What is callback in JavaScript?

Ans: -In JavaScript, a callback is a function that you pass into another function as an argument, and that gets executed at a

later time. It’s like saying, “When you’re done with what you’re doing, do this for me.”

# Simple Explanation

Imagine you ask someone to do a task and then, once they’ve finished, they let you know and might do something else for you. In programming, a callback works similarly. You give a function to another function, and the second function calls back your function when it’s finished doing its work.

html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Callback Example</title>
</head>
<body>
    <button id="myButton">Click me!</button>

    <script>
        // Define the callback function
        function handleClick() {
            alert('Button was clicked!');
        }

        // Get the button element
        const button = document.getElementById('myButton');

        // Attach the callback function to the button's 'click' event
        button.onclick = handleClick;
    </script>
</body>
</html>
```

# How It Works: -

1. Define the Callback Function: handleClick is the function that will be called when the button is clicked. It just shows an alert box with a message.
2. Get the Button Element: document.getElementById('myButton') retrieves the button element from the HTML.
3. Attach the Callback Function: button.onclick = handleClick; assigns the handleClick function as the callback for the button’s click event. This means that when the button is clicked, the handleClick function will be executed.

In this example, the handleClick function is a callback function that runs whenever the button is clicked. The onclick event is the mechanism that ensures the callback function (handleClick) is called at the right time.

---

3) What is a callback hell in JavaScript?

Ans: -

Callback hell in JavaScript refers to a situation where you have so many nested callbacks (functions that are passed as arguments to other functions) that your code becomes difficult to read and manage.
Imagine you have a series of tasks that need to be completed one after another. If each task is defined as a function that takes a callback to be executed after the task is done, you end up with a series of nested functions. This can make the code look like a pyramid or deeply nested structure, which is hard to follow.
For example:

```javascript
doTask1(function(result1) {
    doTask2(result1, function(result2) {
        doTask3(result2, function(result3) {
            doTask4(result3, function(result4) {
                // and so on...
            });
        });
    });
});
```

In this example, each task depends on the result of the previous one, creating deep nesting.
Why it's a problem:

- **Hard to Read:** The more levels of nesting you have, the harder it is to understand what the code is doing.
- **Difficult to Debug:** Finding and fixing errors can be tricky when the code is deeply nested.
- **Maintenance Issues:** Changes to one part of the code can require adjustments in many nested functions.

Solutions to Callback Hell:

1. **Promises:** These allow you to write asynchronous code in a more readable way by chaining `` `.then()` `` and `` `.catch()` `` methods.
2. **Async/Await:** This syntax allows you to write asynchronous code that looks and behaves like synchronous code, making it easier to read and maintain.

By using these techniques, you can flatten out the nesting and make your code more manageable.

---

4) What is NaN in JavaScript?

Ans: -

In JavaScript, `NaN` stands for "Not a Number." It's a special value that represents something that is not a valid number. For example, if you try to do a mathematical operation that doesn't make sense, like dividing zero by zero, JavaScript will return `NaN`.
Here's a simple example:

```javascript
let result = 0 / 0;  // This will be NaN
console.log(result); // This will print "NaN"
```

So, if you see `NaN` in your code, it usually means there was an error in a calculation or an invalid number was used.

---

5) What is the purpose of the "use strict" statement in JavaScript?

Ans: -

The `"use strict"` statement in JavaScript is a way to make your code run in a "strict mode." Strict mode is a stricter version of JavaScript that helps you write more reliable and error-free code.
Here's a simple breakdown of its purpose:

1. **Catch Errors Early**: It helps identify mistakes and errors that might otherwise be silent or difficult to spot. For example, it catches variables that are used without being declared first.
2. **Prevent Dangerous Actions**: It stops you from doing things that are considered bad practice or potentially harmful. For example, it prevents you from using certain reserved keywords or creating global variables unintentionally.
3. **Improve Performance**: Sometimes, strict mode can help JavaScript engines optimize the code better, leading to potentially faster performance.

To use it, you simply add `"use strict";` at the beginning of your JavaScript file or function. Here's an example:

```javascript
"use strict";
var x = 10;  // This will work fine
```

Strict mode helps make your code cleaner and less prone to subtle bugs.
