Closures

1. What is a closure in JavaScript?

A closure is a function that remembers the variables around it, even after the outer function has finished running.

2. Why are closures useful?

Closures let you keep some data private and only accessible within the function

3. What is the output of the following code?

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count);
  };
}
const counter = outer();
counter();
counter();
```

output
1
2

4. What does a closure in JavaScript allow?

Closures allow a function to access variables from an outer function after the outer function has finished

5. What is the output of the following code?

```
function outer() {

  let count = 5;
  return function() {
    console.log(count);
  };
}
const inner = outer();
inner();
```

OUTPUT:
5

6. In closures, where are the variables stored?

closures scope

7. What is the result of the following code?

```
function makeMultiplier(multiplier) {
  return function(x) {
    return x * multiplier;
  };
}
const double = makeMultiplier(2);
console.log(double(5));
```

OUTPUT:
10

8.

Closures allow functions to "remember" variables from their outer scope even after the outer function has finished executing, enabling private variables

8. Explain how closures can be used to create private variables.
9. What is the output of the following code snippet?
10. How do closures help in callback functions? Closures help in callback functions by preserving the surrounding scope's variables, allowing the callback to access and use them even after the outer function has executed.

```
function createCounter() {
  let count = 0;
  return function() {
    count += 1;
    return count;
  };
}
const counter = createCounter();
console.log(counter());
console.log(counter());
console.log(counter());
```

OUTPUT:
1
2
3

1.When an event happens on an element, it starts at the target element and then bubbles up to its parent elements.

Event Bubbling and Capturing

1. What is event bubbling?
2. How can you stop event bubbling in JavaScript? stopPropagation()
3. What is the order of execution in the "capturing" phase? from the outermost down to the target element.
4. What happens if both capturing and bubbling phases are handled for the same event?
5. In event bubbling, which element's event is captured first? Target element

4.the capturing phase executes first (top-down), followed by the bubbling phase (bottom-up).

9.Event Bubbling: The event starts at the target element and propagates upward to the ancestors.
Event Capturing: The event starts from the outermost ancestor and propagates downward to the target element.

6. Which method can be used to stop event propagation during bubbling? stopPropagation()
7. In event capturing, the order of event handling starts from: window
8. Which JavaScript method adds both capturing and bubbling event listeners? addEventListener()
9. Explain the difference between event bubbling and event capturing.
10. How can you prevent default behavior while stopping propagation of an event?
by using event.preventDefault() and event.stopPropagation() together.

THIS Keyword

1. In the global scope, what does this refer to in JavaScript? window
2. What is the value of this inside a regular function? global scope
3. What happens when you use this inside an arrow function? inherits its value from the lexicalscope
4. What will this refer to in the following code snippet?

```
const obj = {
  method: function() {
    console.log(this);
  }
};
obj.method();
```

OUTPUT:
object

5. In a simple function, what does this refer to in non-strict mode? window
6. In an arrow function, what does this refer to? to the surrounding (lexical) scope.
7. What does this refer to in event handlers? element
8. Explain how this behaves in a constructor function. to the newly created instance
9. How does this behave in a class method? Instance
10. What is the value of this in a method passed as a callback function? global object

Call, Apply, and Bind Functions

1. Executes a function with a specific this value and arguments provided one by one

1. What does the call() method do in JavaScript?
2. How is apply() different from call()? apply() passes arguments as an array, while call() passes them individually.
3. What does bind() return?
4. What is the output of the following code?

bind returns a new function with specified this value

```
const obj = { num: 10 };
function add(a, b) {
  return this.num + a + b;
}
const result = add.call(obj, 20, 30);
```

OUTPUT:
60

```
console.log(result);
```

5. What is the difference between call and apply? apply() passes arguments as an array, while call() passes them individually.
6. What does bind return? bind returns a new function with specified this value
7. What will this code output?

```
const person = {
  firstName: "John",
  lastName: "Doe",
};
function greet(greeting) {
  console.log(greeting + " " + this.firstName + " " +
this.lastName);
}
greet.call(person, "Hello");
```

OUTPUT
Hello John Doe

8. What will this code output?

```
const obj = {
  num1: 10,
  num2: 20,
};
function addNumbers(a, b) {
  return this.num1 + this.num2 + a + b;
}
const result = addNumbers.apply(obj, [30, 40]);
console.log(result);
```

OUTPUT:
100

9. Explain how bind() can be used to create a partially applied function.
10. How can call() and apply() be useful in borrowing methods from other objects?

9.bind() can be used to create a partially applied function by pre-setting some arguments, allowing the function to be called later with the remaining arguments.

10.ll() and apply() allow borrowing methods from other objects by setting the this context to the target object, enabling method reuse.