JAVASCRIPT DAY 25th

CHEAT SHEET OF DAY 25th

Closures

- **Definition**: A closure is a function that retains access to its lexical scope, even when the outer function has completed execution.
- **Why Useful**: They enable data encapsulation and allow private variables that only specific functions can access.
- **Access**: Inner functions can access variables from their outer function scope after the outer function finishes running.
- **Storage**: Variables in closures are stored in a special area known as the closure scope.

Example:

- function outer() {

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

Event Bubbling and Capturing

- **Event Bubbling**: Events triggered on an element propagate upward through its parent elements, starting from the target element.
- **Event Capturing**: Events propagate from the root element downward to the target element before bubbling begins.
- **Add Event Listeners**: Use addEventListener(type, listener, useCapture) where useCapture is true for capturing phase and false for bubbling (default).
- **Stop Event Bubbling**: Use event.stopPropagation() to halt event propagation during bubbling.
- **Prevent Default**: Use event.preventDefault() to stop the default behavior of the event.

Example:

```
document.querySelector("#child").addEventListener("click", (e) => {
  console.log("Child clicked!");
e.stopPropagation(); // Prevent bubbling
```

```
});

document.querySelector("#parent").addEventListener("click", () => {
  console.log("Parent clicked!");
});
```

- **Difference**:

  Bubbling: Event moves from the target element to outer ancestors.

  Capturing: Event moves from outer ancestors to the target element.

  this Keyword

- **Global Scope**: In the global scope, this refers to the global object (window in browsers).
- **Regular Function**: Inside a regular function, this depends on how the function is called:

  Called directly: refers to the global object (non-strict mode) or undefined (strict mode).
- **Arrow Function**: this refers to its surrounding (lexical) scope, not its caller.
- **Constructor Function**: this refers to the newly created object.
- **Class Method**: this refers to the class instance.
- **Callback Function**: this may refer to the global object or be undefined (strict mode) unless explicitly bound.
- **Examples**:

```
const obj = {
value: 10,
regularFn: function () { console.log(this.value); }, // `this` -> obj
arrowFn: () => console.log(this.value), // `this` -> global or outer lexical context
};
obj.regularFn(); // 10
obj.arrowFn();   // undefined
```

  Call, Apply, and Bind

- **Call**: Invokes a function with a specific this value and individual arguments.

```
function greet(greeting) {
console.log(`${greeting}, ${this.name}`);
}
greet.call({ name: "Alice" }, "Hello"); // Hello, Alice
```

  Apply: Similar to call, but arguments are passed as an array.

```
greet.apply({ name: "Bob" }, ["Hi"]); // Hi, Bob
```

- **Bind**: Returns a new function with a bound this value and optional preset arguments.

```
const boundGreet = greet.bind({ name: "Charlie" }, "Hey");
boundGreet(); // Hey, Charlie
```

Key Differences:

- **call**: Arguments are passed individually.
- **apply**: Arguments are passed as an array.
- **bind**: Does not invoke the function; instead, it returns a new function.

CODING QUESTIONS

Closures

1. Simple Closure:

```
function createCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}
const counter = createCounter();
console.log(counter()); // Expected output: 1
console.log(counter()); // Expected output: 2
console.log(counter()); // Expected output: 3
```

2. Closure for Private Variables:

```
function person() {
    let name = 'John';
    return {
        getName: function() {
            return name;
        },
        setName: function(newName) {
            name = newName;
        }
```

```
    };
}
const john = person();
console.log(john.getName()); // Expected output: John
john.setName('Doe');
console.log(john.getName()); // Expected output: Doe
```

3. Closure with Asynchronous Code:

```
function delayedGreeting(name) {
    return function() {
        setTimeout(() => {
            console.log(`Hello, ${name}!`);
        }, 1000);
    };
}
const greet = delayedGreeting('Alice');
greet(); // Expected output after 1 second: Hello, Alice!
```

Event Bubbling and Capturing

1. Event Bubbling Example:

```
<div id="outerDiv">
    <button id="innerButton">Click me!</button>
</div>
<script>
document.getElementById('innerButton').addEventListener('click', function() {
    alert('Button clicked!');
});
document.getElementById('outerDiv').addEventListener('click', function() {
    alert('Div clicked!');
});
</script>
```

2. Stop Event Bubbling:

```
<div id="outerDiv">
    <button id="innerButton">Click me!</button>
```

```
</div>
<script>
document.getElementById('innerButton').addEventListener('click', function(event) {
   alert('Button clicked!');
   event.stopPropagation();
});
document.getElementById('outerDiv').addEventListener('click', function() {
   alert('Div clicked!');
});
</script>
```

3. Event Capturing Example:

```
<div id="outerDiv">
   <button id="innerButton">Click me!</button>
</div>
<script>
document.getElementById('outerDiv').addEventListener('click', function() {
   alert('Div clicked!');
}, true); // Use capturing
document.getElementById('innerButton').addEventListener('click', function() {
   alert('Button clicked!');
});
</script>
```

this Keyword

1. Global Scope Example:

```
console.log(this); // Expected output: Window object (in browsers)
```

2. Function Context Example:

```
function showThis() {
   console.log(this);
}
showThis(); // Expected output: Window object (in non-strict mode)
```

3. Method Context Example:

```
const obj = {
```

```
    value: 42,
    showValue: function() {
        console.log(this.value);
    }
};
obj.showValue(); // Expected output: 42
```

4. Arrow Function Context Example:

```
const obj = {
    value: 42,
    showValue: () => {
        console.log(this.value);
    }
};
obj.showValue(); // Expected output: undefined (arrow function does not have its own `this`)
```

5. Event Handler Context Example:

```
<button id="btn">Click me!</button>
<script>
document.getElementById('btn').addEventListener('click', function() {
    console.log(this); // Expected output: The button element
});
</script>
```

Call, Apply, and Bind Functions

1. Using call():

```
const obj = { num: 10 };
function add(a, b) {
    return this.num + a + b;
}
const result = add.call(obj, 20, 30);
console.log(result); // Expected output: 60
```

2. Using apply():

```
const obj = { num: 10 };
function add(a, b) {
```

```
    return this.num + a + b;
}
const result = add.apply(obj, [20, 30]);
console.log(result); // Expected output: 60
```

3. Using bind():

```
const obj = { num: 10 };
function add(a, b) {
    return this.num + a + b;
}
const boundAdd = add.bind(obj);
console.log(boundAdd(20, 30)); // Expected output: 60
```

4. Method Borrowing with call():

```
const obj1 = { num: 10 };
const obj2 = { num: 20 };
function showNum() {
    console.log(this.num);
}
showNum.call(obj1); // Expected output: 10
showNum.call(obj2); // Expected output: 20
```

FAQ'S OF DAY 25th

Closures

1. **What is a closure in JavaScript?**
   A closure is a function that retains access to its outer scope even after the outer function has finished executing.
2. **Why are closures useful?**
   Closures allow for:

- Data encapsulation and privacy.
- Maintaining state between function calls.
- Creating higher-order functions like callbacks and currying.

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 functions to access variables from their outer scope, even after the outer function has returned.

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?**

   Variables are stored in the function's **lexical environment**, which is preserved by the closure.

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. **Explain how closures can be used to create private variables.**

Closures encapsulate variables, making them inaccessible from the outside. For example:

```
function createCounter() {
 let count = 0; // private variable
 return {
  increment: () => ++count,
getCount: () => count,
 };
}
const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.getCount());  // 1
```

9. **What is the output of the following code snippet?**

```
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

10. **How do closures help in callback functions?**

    Closures help callbacks retain access to variables from their defining scope, allowing them to use those variables asynchronously or later.

Event Bubbling and Capturing

1. **What is event bubbling?**

   Event bubbling occurs when an event on a child element propagates upward to its parent elements.

2. **How can you stop event bubbling in JavaScript?**

   Use event.stopPropagation() to stop the event from propagating further.

3. **What is the order of execution in the "capturing" phase?**

   Events are captured from the outermost element to the innermost target element.

4. **What happens if both capturing and bubbling phases are handled for the same event?**

   Both phases execute, but capturing handlers execute first, followed by bubbling handlers.

5. **In event bubbling, which element's event is captured first?**

   The innermost target element captures the event first.

6. **Which method can be used to stop event propagation during bubbling?**

   event.stopPropagation().

7. **In event capturing, the order of event handling starts from:**

   The outermost parent element and moves inward to the target element.

8. **Which JavaScript method adds both capturing and bubbling event listeners?**

   addEventListener with the third argument set to true for capturing.

9. **Explain the difference between event bubbling and event capturing.**

- **Bubbling:** Events propagate from the target element to its parent elements.
- **Capturing:** Events propagate from the parent elements to the target element.

10. **How can you prevent default behavior while stopping propagation of an event?**

    Use event.preventDefault() and event.stopPropagation() together.

This Keyword

1. **In the global scope, what does this refer to in JavaScript?**

   The global object (e.g., window in browsers).

2. **What is the value of this inside a regular function?**

   In non-strict mode: The global object.

   In strict mode: undefined.

3. **What happens when you use this inside an arrow function?**

   this is lexically inherited from the surrounding scope.

4. **What will this refer to in the following code snippet?**

```
const obj = {
value: 42,
 getValue: function() {
 return this.value;
  },
};
console.log(obj.getValue());
```

Output:

42

5. **In a simple function, what does this refer to in non-strict mode?**

   The global object (window).

6. **In an arrow function, what does this refer to?**

   The this value of the surrounding lexical scope.

7. **What does this refer to in event handlers?**

   The element on which the event is triggered.

8. **Explain how this behaves in a constructor function.**

   Inside a constructor, this refers to the new object being created.

9. **How does this behave in a class method?**

   this refers to the instance of the class.

10. **What is the value of this in a method passed as a callback function?**

    By default, this refers to the global object unless explicitly bound.

Call, Apply, and Bind Functions

1. **What does the call() method do in JavaScript?**

   Invokes a function with a specified this context and arguments passed individually.

2. **How is apply() different from call()?**

   apply() accepts arguments as an array, while call() takes them individually.

3. **What does bind() return?**

   A new function with a specific this value and optionally preset arguments.

4. **What is the output of the following code?**

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

```
}
const result = add.call(obj, 20, 30);
console.log(result);
```

Output:

60

5. **What is the difference between call and apply?**

- call: Arguments are passed individually.
- apply: Arguments are passed as an array.

6. **What does bind return?**

A new function with the 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.**

bind() allows presetting some arguments of a function:

```
function add(a, b) {
  return a + b;
}
const addFive = add.bind(null, 5);
console.log(addFive(10)); // 15
```

10. **How can call() and apply() be useful in borrowing methods from other objects?**

Methods from one object can be used with another object's context:

```
const obj1 = { num: 10 };
const obj2 = { num: 20 };
function showNum() {
console.log(this.num);
}
showNum.call(obj1); // 10
showNum.call(obj2); // 20
```

MCQ'S OF DAY 25th

Closures

1. **What is a closure in JavaScript?**

   A. A function combined with its lexical environment.

2. **Why are closures useful?**

   B. To create private variables.

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();
```

B. 1, 2.

4. **What does a closure in JavaScript allow?**

   A. Access to variables within a function after the function execution has completed.

5. **Which of the following will output 5?**

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

   A. 5.

6. **In closures, where are the variables stored?**

   D. Closure 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));
```

   B. 10.

Event Bubbling and Capturing

1. **What is event bubbling?**

   A. Events start from the innermost element and propagate outward.

2. **How can you stop event bubbling in JavaScript?**

   A. event.stopPropagation().

3. **What is the order of execution in the "capturing" phase?**

   A. Outermost to innermost element.

4. **What happens if both capturing and bubbling phases are handled for the same event?**

   C. Both phases are executed in order.

5. **In event bubbling, which element's event is captured first?**

   B. Innermost child.

6. **Which method can be used to stop event propagation during bubbling?**

   C. stopPropagation().

7. **In event capturing, the order of event handling starts from:**

   C. Root to target.

8. **Which JavaScript method adds both capturing and bubbling event listeners?**

   B. addEventListener(type, listener, useCapture).

THIS Keyword

1. **In the global scope, what does this refer to in JavaScript?**

   A. The window object.

2. **What is the value of this inside a regular function?**

   D. All of the above. (Depends on the context and mode, i.e., strict vs non-strict.)

3. **What happens when you use this inside an arrow function?**

   A. It takes the value of this from the enclosing lexical scope.

4. **What will this refer to in the following code snippet?**

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

   C. obj.

5. **In a simple function, what does this refer to in non-strict mode?**

   A. Global object.

6. **In an arrow function, what does this refer to?**

   C. Lexical scope.

7. **What does this refer to in event handlers?**

```
document.getElementById('btn').addEventListener('click', function () {
```

```
console.log(this);
});
```

A. The button element.

Call, Apply, and Bind Functions

1. **What does the call() method do in JavaScript?**

   B. Executes a function with a given this value and arguments provided individually.

2. **How is apply() different from call()?**

   B. It takes arguments as an array.

3. **What does bind() return?**

   A. A new function.

4. **What is the output of the following code?**

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

   C. 60.

5. **What is the difference between call and apply?**

   C. apply accepts arguments as an array, and call takes them separately.

6. **What does bind return?**

   B. A new function bound to a specific object.

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");
```

   B. 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);
```

A. 100.