

Closures

Explanation : A closure is when a function "remembers" variables from its outer scope, even after that outer function has finished executing.

- Enables data privacy and function factories.

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

Promises & Async/Await

Explanation: Promises handle asynchronous operations.
Async/await makes promise code look synchronous.

- Essential for API calls and handling delayed operations.



```
=> Promise
```

```
fetch('https:...')  
  .then(response => response.json())  
  .then(data => console.log(data));
```

```
=> Async/Await
```

```
async function getData() {  
  const response = await fetch('https:...');  
  const data = await response.json();  
  console.log(data);  
}
```

The 'this' Keyword

Explanation: 'this' refers to the object executing the current function. Its value depends on HOW the function is called.



```
const user = {  
  name: 'Shailesh',  
  greet: function() {  
    console.log('Hi, I am ' + this.name);  
  }  
};
```

```
user.greet(); => "Hi, I am Shailesh"
```

```
const greetFunc = user.greet;  
greetFunc(); => "Hi, I am undefined"
```

```
//..... Fix with bind().....
```

```
const boundGreet = user.greet.bind(user);  
boundGreet(); => "Hi, I am Shailesh"
```

Event Loop

Explanation: JavaScript is single-threaded but uses an event loop to handle async tasks without blocking.

- Why it matters: Understand how code execution order works.



```
console.log('First');
```

```
setTimeout(() => {  
  console.log('Second');  
}, 0);
```

```
console.log('Third');
```

=> **Output:** First, Third, Second

=> setTimeout goes to task queue,
runs after main code.

Hoisting

Explanation: Variable and function declarations are moved to the top of their scope during compilation.

= Explains why some code works before declaration.



```
console.log(x); => undefined (not an error)
var x = 5;
```

```
=> var x; (hoisted)
=> console.log(x);
=> x = 5;
```

```
greet(); => "Hello!" - Functions are hoisted
```


```
function greet() {
  console.log('Hello!');
}
```

```
=> BUT let/const are NOT initialized when hoisted
console.log(y); => ReferenceError!
let y = 10;
```

Arrow Functions

Explanation: A shorter syntax for functions. They DON'T have their own 'this'.

- Great for callbacks, but behaves differently than regular functions



```
console.log(x); => undefined (not an error)
var x = 5;
```

```
=> var x; (hoisted)
=> console.log(x);
=> x = 5;
```

```
greet(); => "Hello!" - Functions are hoisted
```

```
function greet() {
  console.log('Hello!');
}
```

```
=> BUT let/const are NOT initialized when hoisted
console.log(y); => ReferenceError!
let y = 10;
```

Destructuring

Explanation: Extract values from arrays or objects into separate
Cleaner, more readable code.



```
..... Array Destructuring .....  
const colors = ['red', 'green', 'blue'];  
const [first, second] = colors;  
console.log(first); => 'red'  
  
..... Object Destructuring .....  
const person = { name: 'John', age: 30, city: 'NYC' };  
const { name, age } = person;  
console.log(name);    => 'John'  
  
..... With function parameters .....  
function printUser({ name, age }) {  
  console.log(`${name} is ${age} years old`);  
}  
printUser(person);
```


Spread & Rest Operators

Explanation: The ... operator. Spread expands arrays/objects. Rest collects items into an array.

– Powerful tool for copying and combining data.



-----Spread - expands elements

```
const arr1 = [1, 2, 3];
```

```
const arr2 = [...arr1, 4, 5]; => [1, 2, 3, 4, 5]
```

```
const user = { name: 'Bob', age: 25 };
```

```
const updatedUser = { ...user, age: 26 };
```

```
=> Copy and update
```

----- Rest - collects into array

```
function sum(...numbers) {
```

```
  return numbers.reduce((total, num) => total + num, 0);
```


```
}
```

```
console.log(sum(1, 2, 3, 4)); => 10
```


map(), filter(), reduce()

Explanation: Array methods for transforming data without mutating the original array.

- Functional programming style, essential for modern JS.



```
const numbers = [1, 2, 3, 4, 5];

----- map - transform each element
const doubled = numbers.map(num => num * 2);
=> [2, 4, 6, 8, 10]


----- filter - keep elements that pass test
const evens = numbers.filter(num => num % 2 === 0);
=> [2, 4]

----- reduce - reduce to single value
const sum = numbers.reduce((total, num) => total + num, 0);
=> 15
```

Call, Apply, and Bind

Explanation: Methods to control what 'this' refers to in a function.

- Crucial for context management in JavaScript.



```
const person = {
  name: 'Sarah',
  greet: function(greeting, punctuation) {
    console.log(`${greeting}, I'm ${this.name}${punctuation}`);
  }
};
```

```
const anotherPerson = { name: 'Mike' };
```

```
---- call - invokes immediately with arguments
person.greet.call(anotherPerson, 'Hello', '!');
=> "Hello, I'm Mike!"
```

```
---- apply - same but arguments as array
person.greet.apply(anotherPerson, ['Hi', '?']);
=> "Hi, I'm Mike?"
```

```
----- bind - returns new function with bound 'this'
const mikeGreet = person.greet.bind(anotherPerson);
mikeGreet('Hey', '!!!');
=> "Hey, I'm Mike!!!"
```