# 10 Advance JavaScript Tricks

# 1. DESTRUCTURING ASSIGNMENT

Assignment destructuring is a concise way to extract values from arrays or objects and assign them to variables.

It simplifies your code and improves readability. For arrays, you can use bracket notation, and you can use braces for objects.

@Dimple Kumari

# 1. DESTRUCTURING ASSIGNMENT

```
// Destructuring arrays
const [firstItem, secondItem, ...rest] =
[1, 2, 3, 4, 5];

// Destructuring objects
const { name, age, ...details } =
{ name: 'John', age: 30, occupation:
'Developer' };
```

@Dimple Kumari

# 2. SPREAD SYNTAX

You can use the spread syntax to extend the elements of an array or the properties of an object into another array or object.

This is useful for making copies, merging objects, and passing multiple arguments to functions.

@Dimple Kumari

# 2. SPREAD SYNTAX

```
// Copy an array
const originalArray = [1, 2, 3];
const newArray = [...originalArray];

// Merge objects
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const mergedObj = { ...obj1, ...obj2 };
```

@Dimple Kumari

# 3. CURRYING

Currying is a functional programming technique in which a function that takes multiple arguments is transformed into a sequence of functions, each taking a single argument.

This allows for better reuse and composition of the code.

@Dimple Kumari

# 3. CURRYING

```
const multiply = (a) => (b) => a * b;
const multiplyByTwo = multiply(2);
const result = multiplyByTwo(5); // Output: 10
```

@Dimple Kumari

# 4. MEMOIZATION

It’s a caching technique used to store the results of expensive function calls and avoid unnecessary recalculations.

It can significantly slow the performance of long-term recursive or consuming functions.

@Dimple Kumari

# 4. MEMOIZATION

```
const memoizedFibonacci = (function () {
    const cache = {};

    return function fib(n) {
        if (n in cache) return cache[n];
        if (n <= 1) return n;

        cache[n] = fib(n - 1) + fib(n - 2);
        return cache[n];
    };
})();
```

@Dimple Kumari

# 5. PROMISES AND ASYNC/AWAIT

Promises and Async/Await are essential to handle asynchronous operations more gracefully and make code more readable and maintainable.

They help avoid callbacks hellish and improve error handling.

@Dimple Kumari

# 5. PROMISES AND ASYNC/AWAIT

```
// Using Promises
function fetchData() {
    return new Promise((resolve, reject) => {
      // Asynchronous operation, e.g., fetching data from an API
      // resolve(data) or reject(error) based on the operation result
    });
  }

  // Using Async/Await
  async function fetchData() {
    try {
      const response = await fetch('https://api.example.com/data');
      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Error fetching data:', error);
      throw error;
    }
  }
```

@Dimple Kumari

# 6. CLOSURES

Closures are functions that remember the environment in which they were created, even if that environment is no longer accessible.

They are useful for creating private variables and for behavior encapsulation.

@Dimple Kumari

# 6. CLOSURES

```
function createCounter() {
    let count = 0;
    return function () {
      return ++count;
    };
}

const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
```

@Dimple Kumari

# 7. FUNCTION COMPOSITION

Function composition is the process of combining two or more functions to create a new function.

It encourages code reuse and helps create transformations complex step by step.

@Dimple Kumari

# 7. FUNCTION COMPOSITION

```
const add = (x) => x + 1;
const multiplyByTwo = (x) => x * 2;
const compose = (...fns) => (x) =>
fns.reduceRight((acc, fn) => fn(acc), x);

const addAndMultiply = compose(multiplyByTwo,
add);
console.log(addAndMultiply(3)); // Output: 8
```

@Dimple Kumari

# 8. PROXY

The proxy object allows you to create custom behavior for basic object operations. It allows you to intercept and modify object operations. ‘object, such as accessing properties, assigning, and calling methods.

@Dimple Kumari

# 8. PROXY

```
const handler = {
    get: (target, prop) => {
        console.log(`Accessing property: $
        {prop}`);
        return target[prop];
    },
};

const targetObj = { name: 'John', age: 30 };
const proxyObj = new Proxy(targetObj, handler);

console.log(proxyObj.name); // Output:
Accessing property: name \n John: 8
```

@Dimple Kumari

# 9. EVENT DELEGATION

Event delegation is a technique in which you attach a single event listener to a parent rather than multiple listeners to each child. memory usage and improves performance, especially for large lists or dynamically generated content.

@Dimple Kumari

# 9. EVENT DELEGATION

```
document.getElementById('parent').
addEventListener('click', function (event) {
    if (event.target.matches('li')) {
        console.log('You clicked on an li
        element!');
    }
});
```

@Dimple Kumari

# 10. WEB WORKERS

Web Workers allow you to run JavaScript code in the background, alongside the main thread.

They are useful for offloading CPU-intensive tasks, Avoid UI hangs and improve performance Responsiveness.

@Dimple Kumari

# 10. WEB WORKERS

```
// In the main thread
const worker = new Worker('worker.js');
worker.postMessage({ data: 'some data' });

// In the worker.js file
self.addEventListener('message', function
(event) {
  const data = event.data;
  // Perform heavy computations with the data
  // Post the result back to the main thread
  self.postMessage({ result: computedResult });
});
```

@Dimple Kumari