Advance Javascript Snippets

EVERY DEVELOPERS MUST KNOW

LokeshkDev

lk_webdev_247

SWIPE

Closures

A function remembers the variables from its outer scope even after the outer function has finished.

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

const counter = outer();
counter(); // 1
counter(); // 2
```


LokeshkDev


lk_webdev_247

SWIPE

Callback Functions

A function passed as an argument and executed later.

```
function greet(name,
callback) {
  callback(`Hello,
${name}`);
}

greet("John", msg =>
console.log(msg));
```

LokeshkDev

lk_webdev_247

SWIPE

Promises

A Promise is used to handle asynchronous operations, like fetching data from an API or reading a file.

👉 It promises to return a result in the future, either a success or a failure.

```
let promise = new
Promise((resolve, reject)
=> {
  // async task
  let success = true;

  if (success) {
    resolve("Task done!");
  } else {
    reject("Task failed.");
  }
});

promise
  .then((result) =>
console.log(result))  //
"Task done!"
  .catch((error) =>
console.log(error)); // if
rejected
```

```
fetch('https://api.example.
com/data')
  .then(response =>
response.json())
  .then(data =>
console.log(data))
  .catch(error =>
console.error('Error:',
error));
```


LokeshkDev


lk_webdev_247

SWIPE

Async/Await

async/await is a cleaner, easier way to handle asynchronous operations, like fetching data — without using .then() and .catch() everywhere.

- **async makes a function return a Promise.**
- **await pauses the function until the Promise resolves or rejects.**

```
// A function to fetch user data using async/await
async function getUser() {
  try {
    const response = await
fetch('https://jsonplaceholder.typicode.com/users/1');
    const user = await response.json();

    console.log("User Name:", user.name);
    console.log("Email:", user.email);
  } catch (error) {
    console.error("Failed to fetch user:", error);
  }
}

getUser();
```

LokeshkDev

lk_webdev_247

SWIPE

Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope (function or global) before code runs.

✅ Only declarations are hoisted, not initializations.

```
var a;
console.log(a);  // undefined
a = 5;
```

```
// Function Hoisting
greet();  // ✅ Works!

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

// Function Declaration
sayHi();  // ❌ TypeError

var sayHi = function() {
  console.log("Hi!");
};
```

LokeshkDev

lk_webdev_247

SWIPE

Temporal Dead Zone (TDZ)

The time between when a variable is hoisted and when it is initialized.

This applies to variables declared with let and const

```
console.log(x); // ❌
ReferenceError: Cannot access
'x' before initialization
let x = 10;
```

TDZ Timeline

```
// TDZ starts
let price = 100;  // 🟢
Initialization happens here
```

LokeshkDev

lk_webdev_247

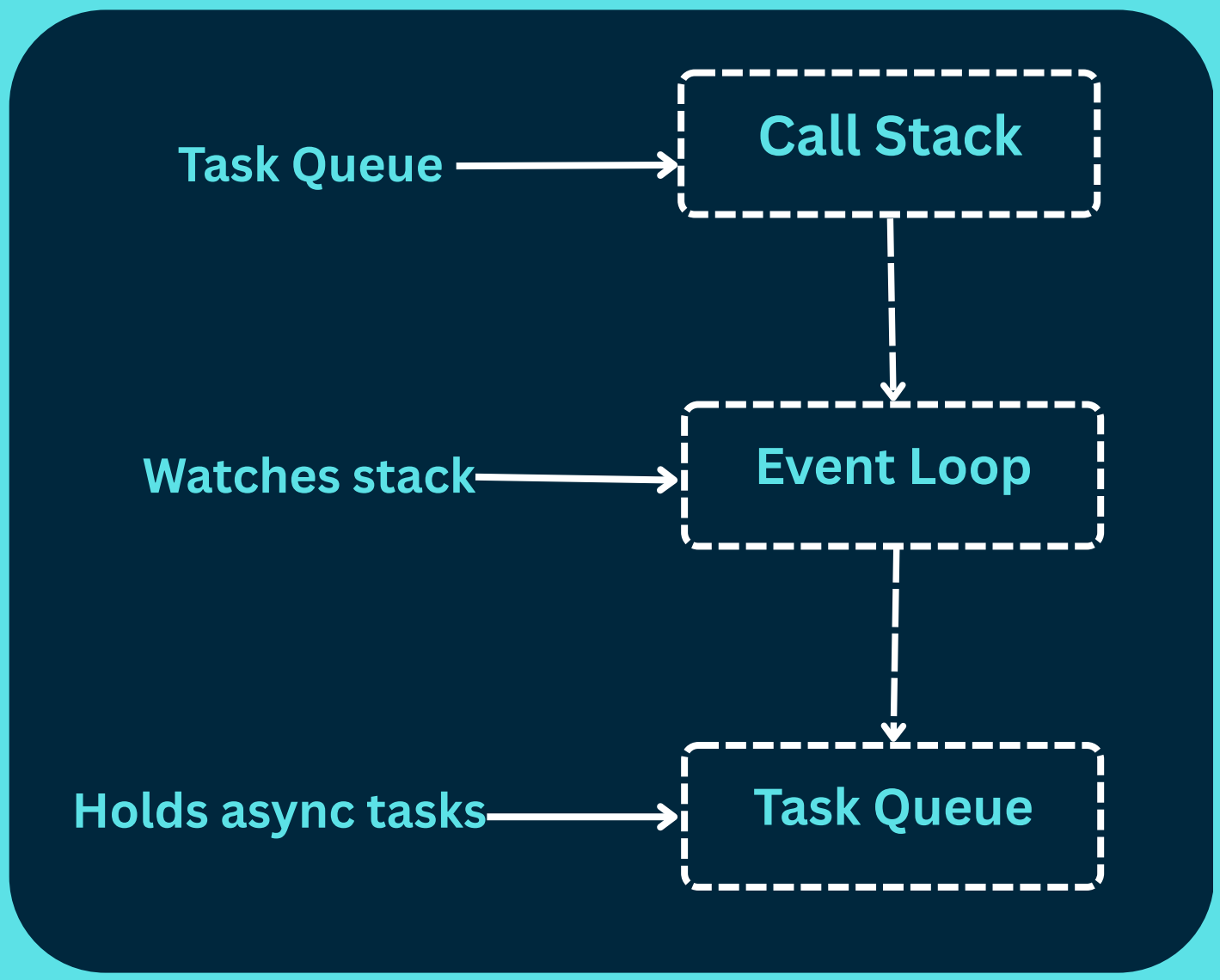
SWIPE

Event Loop + Call Stack

JS handles sync (stack) and async (queue) code in a non-blocking way.

- **The Call Stack runs the code**
- **The Event Loop decides when to run async tasks**
- **The Task Queue holds the waiting callbacks**

How it



LokeshkDev

lk_webdev_247

SWIPE

== VS ===

== (Double Equals) — Loose Equality
Compares values only and converts types if needed (type coercion).

=== (Triple Equals) — Strict Equality
Compares value and type, no type conversion.

```
5 == "5";     // true (number and string are converted to same type)
0 == false;   // true
null == undefined; // true
```

```
5 === "5";     // false (number !== string)
0 === false;   // false
null === undefined; // false
```

Always use === for safer, more predictable comparisons.


LokeshkDev


lk_webdev_247

SWIPE

Currying

Currying turns a function with multiple arguments into a chain of functions, each taking one argument at a time.

Normal Function

- Takes all its arguments at once and returns a result.

```
function add(a, b) {
  return a + b;
}

add(2, 3); // 👉 5
```

Vs

Currying Version

- First call: add(2) → returns a function
- Second call: That returned function is called with 3

```
function add(a) {
  return function(b) {
    return a + b;
  };
}

add(2)(3); // 👉 5
```

LokeshkDev

lk_webdev_247

SWIPE

Debounce & Throttle

Used to optimize performance by controlling how often a function runs — especially during rapid events like typing, scrolling, or resizing.

Debounce delays function until user stops typing/clicking.

```
// Debounce example
function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() =>
fn(...args), delay);
  };
}
```

Throttle limits execution to once every X ms.

```
function throttle(func, limit) {
  let lastCall = 0;
  return function () {
    const now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      func();
    }
  };
}
```

LokeshkDev

lk_webdev_247

SWIPE

THANK YOU SO MUCH!

Follow me on more

LokeshkDev

lokesh-v-kumar

lk_webdev_247