Documentation Tool - JSDoc

Introduction to JSDoc:

JSDoc is a tool for generating documentation directly from JavaScript code. By adding structured comments within the code, it is possible to automatically produce clear, navigable API documentation, usually in HTML format.

This approach ensures that the codebase stays self-documenting and improves support in development environments like VS Code, enabling features such as IntelliSense, autocompletion, and basic type checking

Basic Syntax and Structure:

JSDoc comments follow a specific multi-line format and should be placed immediately before the code element they describe, such as a function, class, module, or variable.

A JSDoc comment block begins with /** and ends with */. Inside the block, it is possible to include a description, parameter details, return types, and other metadata.

Lab2 > doc > JS bb > ...

```
1   /**
2    * Returns a greeting message for a given user.
3    * @param {string} name - The user's name.
4    * @returns {string} A greeting message.
5    */
6   function greetUser(name) {
7       return "Hello, " + name + "!";
8   }
9
```

Here in the given example, we can see that a function “greetUser” is returning a greeting message to the user. The comment above the code is written using the tags of JSDoc such as @param , @returns.

Tags and Types:

JSDoc provides a range of tags to document different aspects of JavaScript code. These tags make the code easier to understand, maintain, and integrate into automated documentation systems. Below are some commonly used tags, grouped by their purpose, along with examples for each.

Function & Parameter Related Tags

These tags are used to describe functions, their parameters, return values, and other behaviors.

- **@param {type} name** – Documents a function parameter.

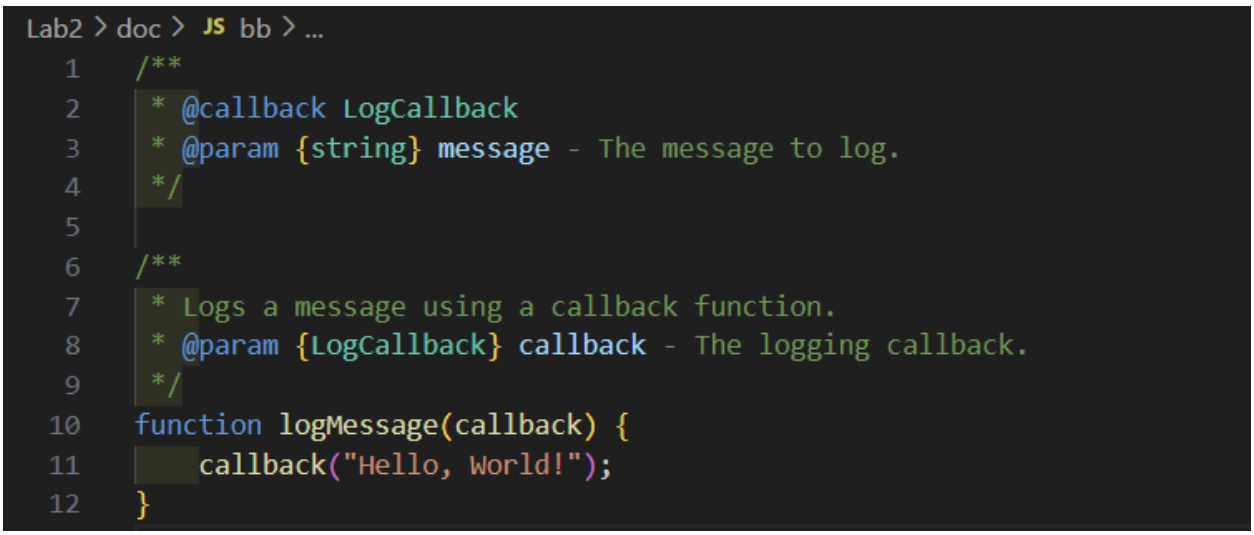
```
Lab2 > doc > JS bb > ...
1   /**
2    * Adds two numbers together.
3    * @param {number} a - The first number.
4    * @param {number} b - The second number.
5    * @returns {number} The sum of both numbers.
6    */
7   function add(a, b) {
8       return a + b;
9   }
10
```

- **@throws {type}** – Describes an error or exception that a function may throw.

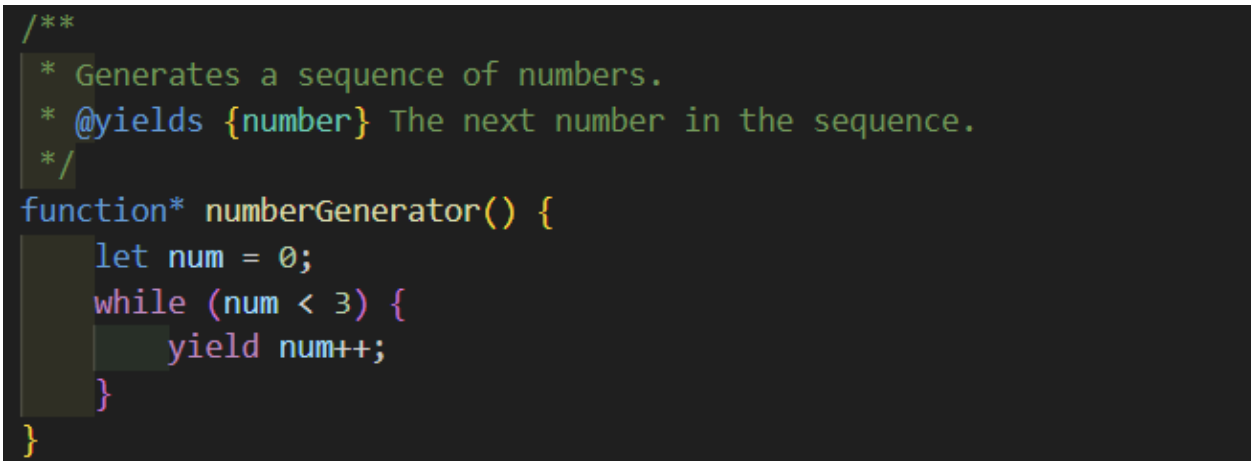
```
Lab2 > doc > JS bb > ...
1   /**
2    * Divides one number by another.
3    * @throws {Error} If the divisor is zero.
4    */
5   function divide(a, b) {
6       if (b === 0) {
7           throw new Error("Cannot divide by zero.");
8       }
9       return a / b;
10  }
```

- **@async** – Marks a function as asynchronous (returns a Promise).

- **@callback** – Describes a callback function type.



- **@yields {type}** – Documents values yielded by a generator function.



- **@returns {type}** – Describes the return value of a function.

```
Lab2 > doc > JS bb > ...
1   /**
2    * Retrieves the current temperature.
3    * @returns {number} The current temperature in Celsius.
4    */
5   function getTemperature() {
6       return 25;
7   }
8
```

Variable & Type Related Tags

These tags are used to define or describe variables, constants, and custom data types.

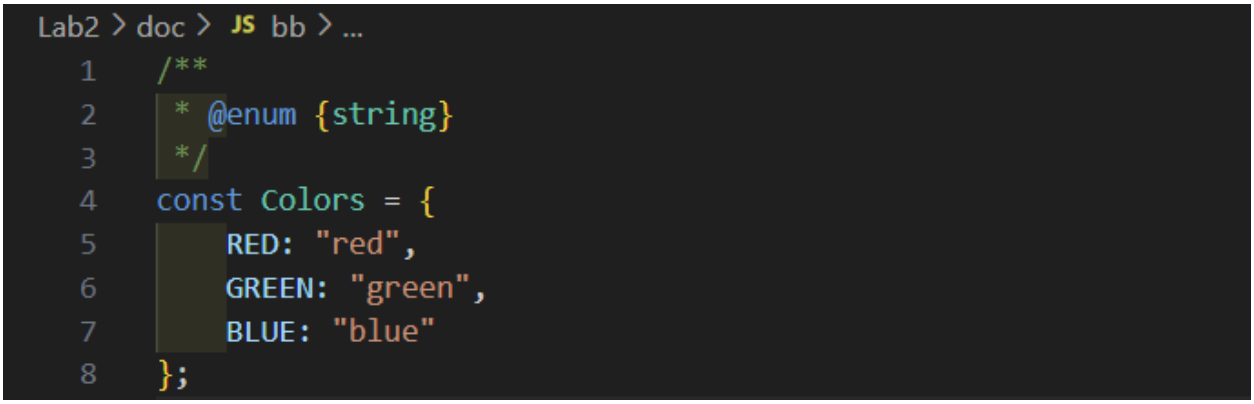
- **@type {type}** – Specifies the data type of a variable or property.

```
Lab2 > doc > JS bb > ...
1   /**
2    * @type {string}
3    */
4   let username = "Mrittika";
5
```

- **@const** – Marks a variable as constant.

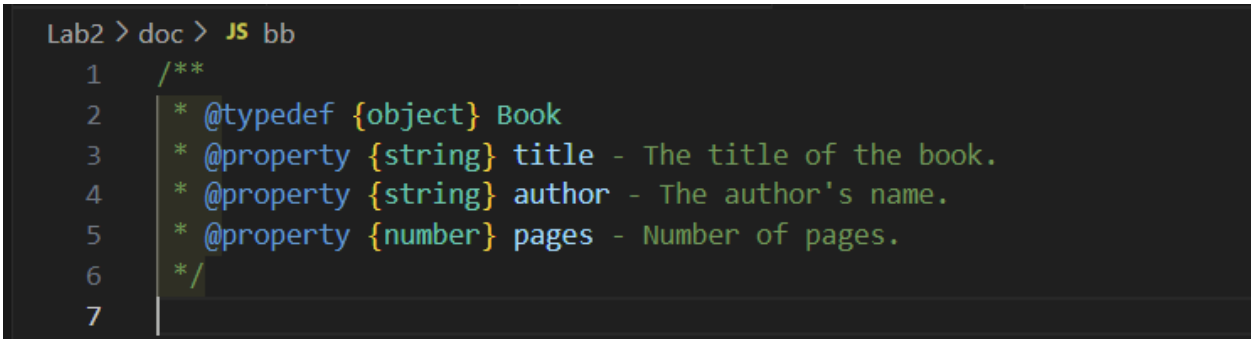
```
Lab2 > doc > JS bb > ...
1   /**
2    * @const {number}
3    * @default
4    */
5   const MAX_USERS = 100;
6
```

- **@enum {type}** – Documents a set of related constant values.



- **@typedef {type}** – Defines a custom type alias.

- **@property {type} name** – Documents a property within an object.



Class & Object Related Tags

These tags describe classes, constructors, and object-oriented features such as inheritance and method overriding.

- **@class** – Marks a function or definition as a class.

- **@constructor** – Marks a function as a constructor.

- **@this {type}** – Specifies the type of this within a function.

- **@extends {class}** – Indicates that a class inherits from another class.

```
Lab2 > doc > JS bb > ...
1   /**
2    * @extends Vehicle
3    */
4   class Car extends Vehicle {
5       constructor(model) {
6           super();
7           this.model = model;
8       }
9   }
10
```

- **@override** – Marks a method as overriding a superclass method.

```
Lab2 > doc > JS bb > ...
1   class Animal {
2       speak() {
3           console.log("Animal sound");
4       }
5   }
6
7   class Dog extends Animal {
8       /**
9        * @override
10       */
11      speak() {
12          console.log("Bark!");
13      }
14  }
```

- **@implements {interface}** – Indicates that a class implements an interface.

```
Lab2 > doc > JS bb > ...
1   /**
2    * @interface
3    */
4   function Drivable() {}
5   Drivable.prototype.drive = function() {};
6   /**
7    * @implements {Drivable}
8    */
9   class Truck {
10      drive() {
11          console.log("The truck is moving");
12      }
13  }
```

Installation:

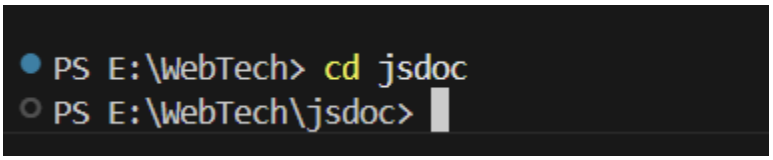
JSDoc can be installed either locally within a project or globally on a system. Installing locally is generally recommended to ensure consistent versions across all developers working on the project.

- **Local (recommended):**
  npm install --save-dev jsdoc
  This ensures that the project always uses a consistent version of JSDoc.
- **Global:**
  npm install -g jsdoc

Installation steps:

Step 1: Open the Project Directory

Navigate to the project folder where documentation will be created.



Step 2: Initialize the Project

Set up a new Node.js project by running the initialization command inside the project folder:

Step 3: Install JSDoc

Add JSDoc as a development dependency to the project:

```
PS E:\WebTech\jsdoc> npm install jsdoc --save-dev

added 30 packages, and audited 31 packages in 6s

2 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS E:\WebTech\jsdoc>
```

The --save-dev flag ensures that JSDoc is installed only for development purposes, not for production builds.

Step 4: Generate the Documentation

Once JSDoc is installed, documentation can be generated from any JavaScript file by running:

```
PS E:\WebTech\jsdoc> npx jsdoc draft.js
PS E:\WebTech\jsdoc>
```

This command scans the file for JSDoc comments and creates documentation files automatically.

Step 5: View the Generated Documentation

After the documentation is generated, a new folder named out/ will appear in the project directory.

Open the generated documentation by locating and opening:

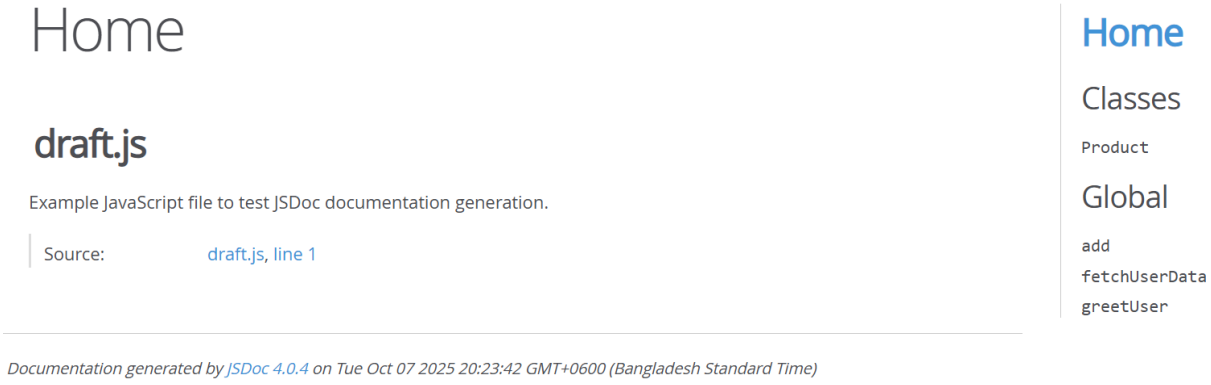
out/index.html

Video Reference:

For a clearer understanding of how JSDoc works, refer to the following tutorial on YouTube:

JSDoc Installation and Usage Guide

Generated Documentation output:



Home

draft.js

Example JavaScript file to test JSDoc documentation generation.

Source: draft.js, line 1

Home

Classes

Product

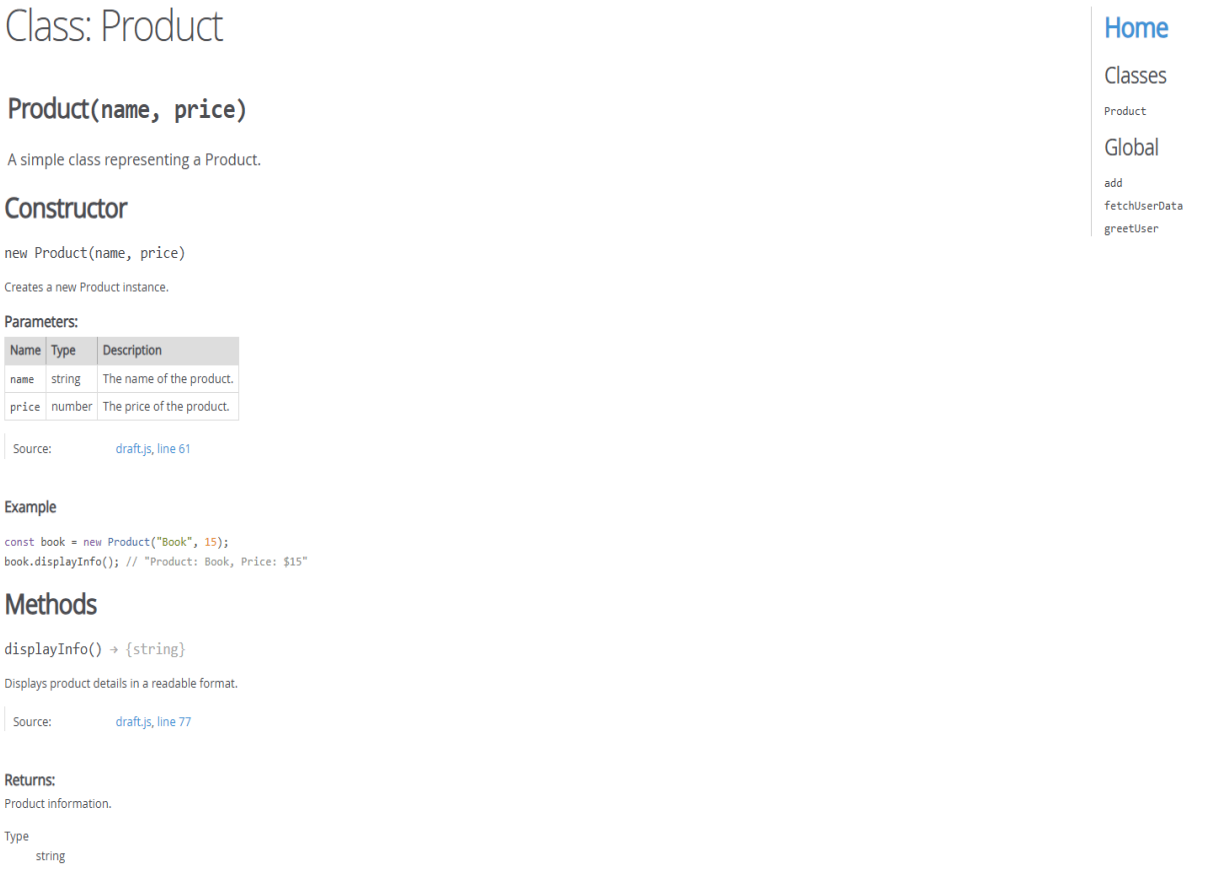
Global

add

fetchUserData

greetUser

Documentation generated by JSDoc 4.0.4 on Tue Oct 07 2025 20:23:42 GMT+0600 (Bangladesh Standard Time)



Class: Product

Product(name, price)

A simple class representing a Product.

Constructor

new Product(name, price)

Creates a new Product instance.

Parameters:

Name	Type	Description
name	string	The name of the product.
price	number	The price of the product.

Source: draft.js, line 61

Example

```
const book = new Product("Book", 15);
book.displayInfo(); // "Product: Book, Price: $15"
```

Methods

displayInfo() → {string}

Displays product details in a readable format.

Source: draft.js, line 77

Returns:

Product information.

Type

string

Home

Classes

Product

Global

add

fetchUserData

greetUser

add
fetchUserData
greetUser

add(a, b) → {number}

Calculates the sum of two numbers.

Parameters:

Name	Type	Description
a	number	The first number.
b	number	The second number.

Source: draft.js, line 35

Returns:

The result of adding the two numbers.

Type
number

Example

```
const total = add(5, 10);
console.log(total); // 15
```

(async) fetchUserData() → {Promise.<User>}

Fetches user data from a mock API.

Source: draft.js, line 46

Returns:

A promise that resolves with user data.

Type
Promise.<User>

Example

```
fetchUserData().then(user => console.log(user.name));
```

greetUser(name) → {string}

Greets a user with a personalized message.

(async) fetchUserData() → {Promise.<User>}

Fetches user data from a mock API.

Source: draft.js, line 46

Returns:

A promise that resolves with user data.

Type
Promise.<User>

Example

```
fetchUserData().then(user => console.log(user.name));
```

greetUser(name) → {string}

Greets a user with a personalized message.

Parameters:

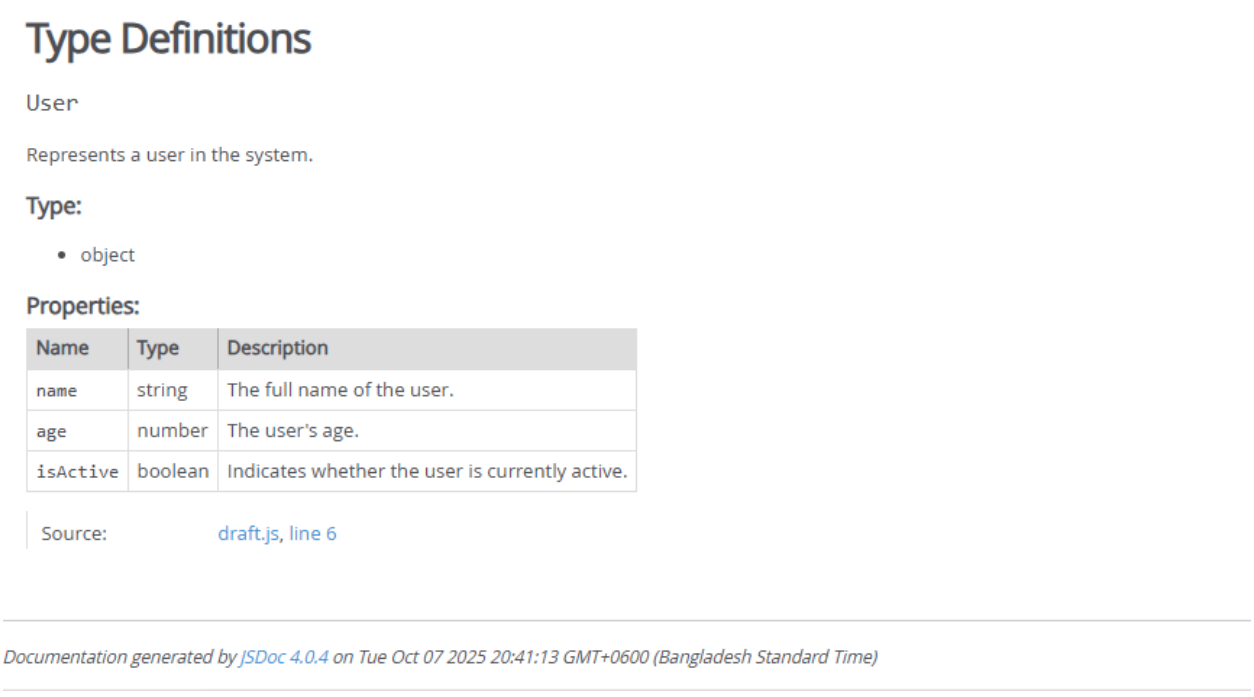
Name	Type	Description
name	string	The user's name.

Source: draft.js, line 22

Returns:

A friendly greeting message.

Type
string



Type Definitions

User

Represents a user in the system.

Type:

- object

Properties:

Name	Type	Description
`name`	string	The full name of the user.
`age`	number	The user's age.
`isActive`	boolean	Indicates whether the user is currently active.

Source: draft.js, line 6

Documentation generated by JSDoc 4.0.4 on Tue Oct 07 2025 20:41:13 GMT+0600 (Bangladesh Standard Time)

Advantages of JSDoc:

1. **Enhances Code Clarity**
   JSDoc helps make your code much easier to understand. By clearly explaining the purpose of functions, parameters, and return values, it ensures that anyone reading your code—whether it's a teammate or your future self—can quickly grasp what each part does.
2. **Improves IDE Functionality**
   Many modern code editors, such as VS Code and WebStorm, can use JSDoc comments to provide helpful features like auto-completion, type hints, and tooltips. This reduces errors and speeds up development by giving real-time guidance while coding.
3. **Facilitates Automatic Documentation**
   With JSDoc, you can automatically generate professional-looking documentation in HTML format. This eliminates the need to manually write separate documentation, saving time and keeping your docs in sync with the code.

4. **Provides Basic Type Safety**
   Even without TypeScript, JSDoc allows you to specify types for variables and function parameters. This lightweight type checking helps catch mistakes early and improves the reliability of your code.

Disadvantages of JSDoc:

1. **Requires Extra Effort**
   Writing comprehensive JSDoc comments can be time-consuming, especially for large projects. Maintaining these comments as the code evolves adds additional workload for developers.
2. **Risk of Becoming Outdated**
   If developers forget to update the comments after changing the code, the documentation can become misleading. Outdated JSDoc may cause confusion rather than help.
3. **Limited Type Enforcement**
   While JSDoc can suggest types, it doesn't enforce them strictly like TypeScript does. Runtime errors may still occur, so it's not a substitute for a fully typed language.
4. **Learning Curve for New Developers**
   Beginners might find the JSDoc syntax and tags like @typedef, @param, and @returns a bit confusing at first. Understanding how to use them effectively can take time.

JSDoc is a valuable tool for documenting JavaScript code and improving code readability, IDE support, and type hinting. However, it requires discipline to maintain, and it cannot fully replace strict type enforcement. Using JSDoc thoughtfully can enhance development efficiency without adding unnecessary overhead.