Introduction to JSDoc

JSDoc is an open-source documentation generator made for JavaScript. It parses specially formatted comments (docblocks) in your source code and produces structured documentation (HTML, JSON, or Markdown). Using JSDoc keeps documentation close to code and lets you automatically generate human-readable API docs that stay synchronized with your implementation.

install & run

Install locally (recommended):

```
npm install --save-dev jsdoc
```

Or install globally:

```
npm install -g jsdoc
```

Basic run:

```
npx jsdoc -r src -d docs

./node_modules/.bin/jsdoc -c jsdoc.json
```

Add npm script in package.json:

```
"scripts": {
  "docs": "jsdoc -c jsdoc.json"
}
```

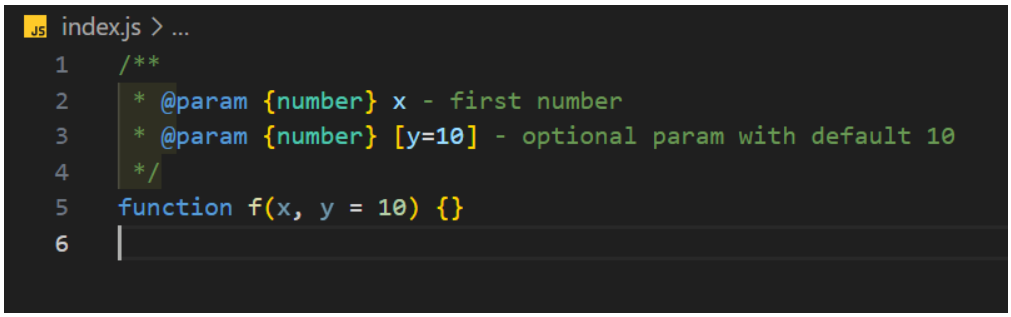
Core tags

Tag	Description
@param	Describes a function parameter.
@returns / @return	Specifies what the function returns.
@example	Shows an example of how to use the code.
@typedef	Defines a custom data type or object structure.
@property	Describes a property inside an object or typedef.

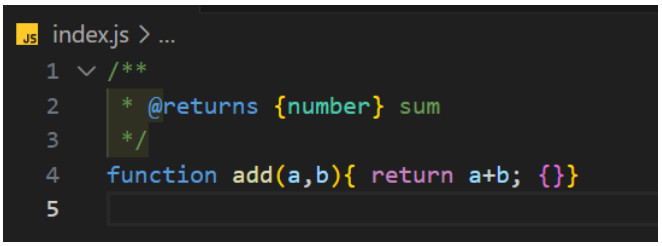
@class	Marks a function or declaration as a class.
@constructor	Indicates that a function is a constructor.
@extends	States that a class inherits from another class.
@implements	Declares that a class implements an interface.
@module	Defines a JavaScript module.
@exports	Marks what a module exports.
@namespace	Groups related functions or variables.
@memberof	Indicates membership of a class or namespace.
@private	Marks an item as private (not for public use).
@protected	Marks an item as protected (for subclasses).
@public	Marks an item as publicly accessible.
@deprecated	Indicates that something should no longer be used.
@since	Specifies when a feature was added.
@throws / @exception	Documents possible errors a function can throw.
@see	Adds a reference or link to related information.
@file	Describes the purpose of a JavaScript file.
@description	Gives a detailed explanation of a function or element.
@author	Names the author of the code.
@version	Indicates the version of the function or module.
@async	Marks a function as asynchronous.
@callback	Defines the signature of a callback function.
@template	Declares a generic type parameter.
@example	Provides sample usage of the code.
@link / {@link}	Creates a clickable link to another symbol or URL.

Examples:

@param



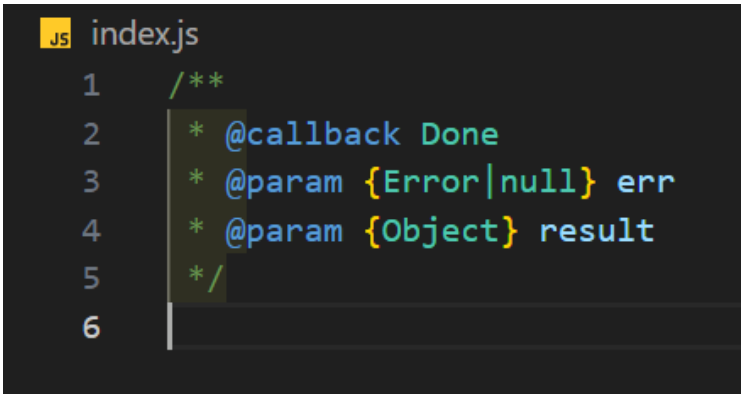
@returns / @return



@example

@typedef + @property

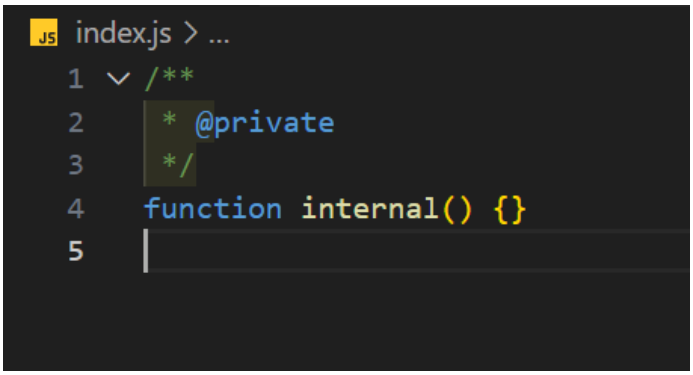
@callback



@class, @constructor, @extends, @implements

@module, @exports, @memberof

@private, @protected, @public, @deprecated, @since



@throws

linking tags

- {@link URL_or_symbol} — inline links inside descriptions.
- @see — references.

Types & type expressions

- Primitives: string, number, boolean
- Array: string[] or Array.<string> or Array<string>
- Union: {string|number}
- Nullable: {?string} or {string|null}
- Record/object map: Object.<string, number> or {[key: string]: number}
- Promise: Promise.<Array.<number>> or Promise<number[]>
- Function type: {function(string,number):boolean} or {(a:number,b:number)=>boolean}
- Rest param: ...number (e.g. @param {...number} nums)

Practical examples

Simple function

index.js > ...

```
 * Multiply two numbers.
 * @param {number} x - First number.
 * @param {number} y - Second number.
 * @returns {number} Product of x and y.
 * @example
 * multiply(2, 3); // 6
 */
function multiply(x, y) { return x * y; }
```

Options object + typedef

index.js > ...

```

/**
 * Fetch data from server.
 * @param {string} url
 * @param {FetchOptions} [opts={}] - options
 * @returns {Promise<Object>}
 */
async function fetchData(url, opts = {}) {}
```

Class

index.js > User > greet

```
/**
 * Represents a User.
 * @class
 */
class User {
/**
   * @param {string} name
   */
constructor(name) {
   /** @type {string} */
   this.name = name;
}

/**
   * Get user greeting.
   * @returns {string}
   */
greet() { return `Hi ${this.name}`; }
}
```

Async + Promise

index.js > ...

```
/**
 * @async
 * @param {string} url
 * @returns {Promise<Object>} parsed JSON
 */
async function getJson(url) {}
```

React functional component (JSX)

index.js > ...

```
/**
 * Button props
 * @typedef {Object} ButtonProps
 * @property {string} label
 * @property {() => void} [onClick]
 */

/**
 * Button component.
 * @param {ButtonProps} props
 * @returns {JSX.Element}
 */
export function Button({label, onClick}) { return <button onClick={onClick}>{label}</button>; }
```

jsdoc.json configuration (example)

index.js | jsdoc.json

jsdoc.json > ...

```
{
  "tags": { "allowUnknownTags": true },
  "source": {
    "include": ["src", "README.md"],
    "includePattern": ".+\\.js(doc|x)?$",
    "excludePattern": "(^|\\/|\\\\)_"
  },
  "plugins": ["plugins/markdown"],
  "templates": {
    "cleverLinks": false,
    "monospaceLinks": false
  },
  "opts": {
    "encoding": "utf8",
    "destination": "./docs",
    "recurse": true,
    "template": "node_modules/minami"
  }
}
```

Advantages and Disadvantages of JSDoc:

Advantages:

Self-Documentation:

Advantage: JSDoc allows developers to embed documentation directly within the code. This makes the code self-documenting, providing details about functions, parameters, return types, and more.

Why it matters: Self-documenting code improves code readability and helps developers understand the purpose and usage of various elements without referring to external documentation.

Automated Documentation Generation:

Advantage: JSDoc facilitates the automatic generation of documentation using tools like the JSDoc command-line interface. This ensures that documentation stays up-to-date with the codebase.

Why it matters: Automated documentation saves time and reduces the risk of inconsistencies between code and documentation. It also encourages developers to keep documentation current.

IDE Integration:

Advantage: Many integrated development environments (IDEs), such as Visual Studio Code, support JSDoc. This integration provides features like autocompletion, inline documentation, and tooltips based on the JSDoc comments.

Why it matters: IDE support enhances the development experience, making it easier for developers to explore and use APIs while writing code.

Type Checking and IntelliSense:

Advantage: JSDoc includes support for specifying data types using @type tags. This information can be used by tools for static type checking and to improve IntelliSense features in IDEs.

Why it matters: Type annotations help catch potential errors early in the development process and improve the accuracy of autocompletion suggestions.

Disadvantages:

Overhead in Code Maintenance:

Disadvantage: Writing and maintaining JSDoc comments requires additional effort, especially in large codebases. Developers need to ensure that comments accurately reflect changes to the code.

Consideration: The benefits of self-documenting code need to be weighed against the time and effort spent on maintaining the documentation.

Potential for Outdated Documentation:

Disadvantage: If developers do not update JSDoc comments alongside code changes, the generated documentation may become outdated, leading to confusion.

Consideration: Establishing a process to review and update documentation during code reviews can help mitigate this risk.

Learning Curve for New Developers:

Disadvantage: For developers unfamiliar with JSDoc syntax and conventions, there may be a learning curve to effectively use and understand the documentation.

Consideration: Providing documentation or training on JSDoc usage within the development team can help new members get up to speed.

Potential for Redundancy:

Disadvantage: In some cases, JSDoc comments may duplicate information already present in the code, potentially leading to redundancy.

Consideration: Striking a balance between concise code and comprehensive documentation is important. Developers should avoid unnecessary repetition.

In summary, while JSDoc offers several advantages in terms of code documentation and automation, its effective use requires careful consideration of the potential drawbacks. Balancing the benefits and costs ensures that JSDoc enhances the development process rather than introducing unnecessary complexity.