INTRODUCTION

About JavaScript

JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard. It has dynamic typing, prototype-based object-orientation, and first-class functions. It is a multi-paradigm, supporting event-driven, functional, and imperative programming styles. It has application programming interfaces (APIs) for working with text, dates, regular expressions, standard data structures, and the Document Object Model (DOM).

The ECMAScript standard does not include any input/output (I/O), such as networking, storage, or graphics facilities. In practice, the web browser or other runtime system provides JavaScript APIs for I/O.

JavaScript contains a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects, for example:

- **Client-side JavaScript** extends the core language by supplying objects to control a browser and its *Document Object Model* (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.

- **Server-side JavaScript** extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

This means that in the browser, JavaScript can change the way the webpage (DOM) looks. And, likewise, Node.js JavaScript on the server can respond to custom requests sent by code executed in the browser.

Although Java and JavaScript are similar in name, syntax, and respective standard libraries, the two languages are distinct and differ greatly in design.

JavaScript	Java
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types are not declared (dynamic typing, loosely typed).	Variable data types must be declared (static typing, strongly typed).
Cannot automatically write to a hard disk.	Can automatically write to a hard disk.

Adding JavaScript into an HTML document

The <script> Tag

In HTML, JavaScript code is inserted between <script> and </script> tags.
Code can be written **inline**, or loaded from an **external** file if you want to run the same script on several pages without writing the script on each and every page.
Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

JavaScript in <head> section

Scripts to be **executed when** they are **called,** or when an **event is triggered**, go in the head section. When you place a script in the head section, you will ensure that the script is loaded before anyone uses it. So, scripts that contain functions go in the head section of the document. Then we can be sure that the script is loaded before the function is called. Head scripts start **loading** really **early**, **before** the **DOM** gets to main processing; you want libraries here so they have time to get going.

For example: Inline script in `<head>` section

```
<html>
<head>
```

```
<script type="text/javascript">
   some statements
</script>
</head>
```

JavaScript in <body> section

Scripts to be executed when the page loads go in the body section. When you place a script in the body section it generates the content of the page.
Body scripts are **loaded while** the **DOM is building**. But there is NO guarantee the DOM will finish before your scripts run, even placed at </body> because pages load asynchronously. If you don't want your script to be placed inside a function, or if your script should write page content, it should be placed in the body section.

For example: External script in `<body>` section

```
<html>
<head>
</head>
<body>
     <script src="xxx.js"></script>
</body>
</html>
```

async, defer

In modern websites, scripts are often "heavier" than HTML: their download size is larger, and the processing time is also longer.

When the browser loads HTML and comes across a `<script>...</script>` tag, it can't continue building the DOM. It must execute the script right now. The same happens for external scripts `<script src="..."></script>`: **the browser must wait for the script to download, execute the downloaded script, and only then can it process the rest of the page.**

That leads to two important issues:

1. Scripts can't see DOM elements below them, so they can't add handlers etc.

2. If there's a bulky script at the top of the page, it "blocks the page". Users can't see the page content till it downloads and runs

There are some workarounds to that. For instance, we can put a script at the bottom of the page. But this solution is far from perfect. For example, the browser notices the script (and can start downloading it) only after it downloaded the full HTML document. For long HTML documents, that may be a noticeable delay.
Luckily, there are two `<script>` attributes that solve the problem for us: `defer` and `async`.

defer

The defer attribute **tells the browser not to wait for the script**. Instead, the browser will continue to process the HTML, build DOM. The **script loads "in the background", and then runs when the DOM is fully built**.

In other words:

- Scripts with defer never block the page.
- Scripts with defer always execute when the DOM is ready (but before DOMContentLoaded event).

The defer attribute is only for external scripts.

async

The async attribute means that a script is completely independent:

- The browser doesn't block async scripts (like defer).
- Other scripts don't wait for async scripts, and async scripts don't wait for them.
- DOMContentLoaded and async scripts don't wait for each other:
    - DOMContentLoaded may happen both before an async script (if an async script finishes loading after the page is complete)
    - ...or after an async script (if an async script is short or was in HTTP-cache)

In other words, async **scripts load in the background and run when ready**. **The DOM and other scripts don't wait for them, and async scripts don't wait for them.** A fully independent script that runs when loaded.
The async attribute is only for external scripts like defer.

Summary
Both async and defer have one common thing: downloading of such scripts doesn't block page rendering. But there are also essential differences between them:

	Order	**DOMContentLoaded**
async	*Load-first order*. Their document order doesn't matter – which loads first runs first	Irrelevant. May load and execute while the document has not yet been fully downloaded. That happens if scripts are small or cached, and the document is long enough.
defer	*Document order* (as they go in the document).	Execute after the document is loaded and parsed (they wait if needed), right before DOMContentLoaded.

In practice, defer is used for scripts that need the whole DOM and/or their relative execution order is important. And async is used for independent scripts, like counters or ads. And their relative execution order does not matter.