EDUFY

# The load event: when is the page ready for your JavaScript?

Maximilian Bahrami · 10 November 2026

Run your code at the right moment: load versus DOMContentLoaded, window.onload versus addEventListener, defer and async scripts, document.readyState, and the mistakes that cause "null" errors.

The browser reads your page from top to bottom, and it runs each script the moment it reaches it. Run your code too early and the button you're looking for doesn't exist yet. Run it too late and the page feels slow. By the end of this lesson you'll know which "ready" moment your code needs, and how to pick it.

## The starting code

Here is a complete page whose only job is to say when it has finished loading. Press **Run** to try it: the message appears under the code, just as it would in the browser's console.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Document</title>
</head>
<body>
<script>
// load fires when the whole page has loaded, including
// stylesheets, images and other resources it depends on.
window.addEventListener("load", function () {
console.log("Page is fully loaded");
});
</script>
</body>
</html>
```

Three things are happening here:

- window is the browser tab your page lives in. Events about the whole page, like load, happen on it.
- addEventListener("load", ...) doesn't run the function now. It hands the function to the browser, which calls it later, when load happens.

- console.log prints to the console. In your own browser, open it with **F12**, or **Cmd + Option + J** on a Mac in Chrome.

**Good to know**

> In VS Code, open the file with the Live Server extension (or just double-click it), then open the console. You'll see "Page is fully loaded" once every image and stylesheet has arrived.

## load and DOMContentLoaded

A page becomes "ready" twice:

1. **DOMContentLoaded** fires on document as soon as the HTML has been read and turned into elements (the DOM). Images may still be downloading.
2. **load** fires on window when everything has finished: images, stylesheets, scripts and iframes.

Run this and watch the order:

```
<!DOCTYPE html>
<html lang="en">
<body>
<img id="photo" src="https://picsum.photos/1200/600" alt="A photo" width="300" />
<script>
const photo = document.getElementById("photo");

document.addEventListener("DOMContentLoaded", () => {
console.log("DOM ready. Image width:", photo.naturalWidth);
});

window.addEventListener("load", () => {
console.log("All loaded. Image width:", photo.naturalWidth);
});
</script>
</body>
</html>
```

naturalWidth is the image's real width in pixels. At DOMContentLoaded it is usually **0**, because the picture hasn't downloaded yet. At load it is **1200**.

| Event | Listen on | Fires when | Use it for |
|---|---|---|---|
| DOMContentLoaded | document | the HTML is read and the elements exist | finding elements, adding click handlers, starting your app |

| Event | Listen on | Fires when | Use it for |
|---|---|---|---|
| load | window | every image, stylesheet and script has loaded | image sizes, hiding a loading screen, anything that needs every file |

**Tip**

> Most code only needs the elements, not the pictures. Wait for DOMContentLoaded (or use defer, below) so your page responds sooner. Keep load for the rare code that needs every file.

## window.onload or addEventListener?

You'll see both in other people's code. They differ in one important way:

```
window.onload = function () {
console.log("first");
};
window.onload = function () {
console.log("second");
};

window.addEventListener("load", () => console.log("A"));
window.addEventListener("load", () => console.log("B"));
```

Only "second" is printed from the onload lines: window.onload is a single slot, and assigning to it again replaces the first function. addEventListener adds to a list, so both A and B run, in the order you added them.

Use addEventListener. Two scripts on the same page (yours and a library's, say) can then both wait for load without one silently erasing the other. It also lets you remove a listener with removeEventListener, or run it only once with { once: true }.

## Where the script goes: defer and async

Here is the classic bug. The script is in <head>, so it runs before the browser has read the <body>:

```
<!DOCTYPE html>
<html lang="en">
<head>
<script>
const button = document.querySelector("button");
console.log("Found:", button);
button.addEventListener("click", () => console.log("Clicked"));
```

```
</script>
</head>
<body>
<button>Click me</button>
</body>
</html>
```

querySelector finds nothing and returns null, and then calling addEventListener on null throws a **TypeError**. There are three ways to fix it:

1. Put the <script> at the end of <body>, as in the starting code.
2. Wrap the code in a DOMContentLoaded listener.
3. Keep the script in <head> in its own file, with defer. This is what most modern sites do.

```
<head>
<script src="app.js" defer></script>
</head>
```

A script in its own file can be loaded three ways:

| Script tag | Downloads | Runs | Keeps the order of scripts? |
|---|---|---|---|
| <script src="app.js"> | stops the page while it downloads | at once | yes |
| <script src="app.js" defer> | alongside the page | after the HTML is read, just before DOMContentLoaded | yes |
| <script src="app.js" async> | alongside the page | as soon as it has downloaded | no |
| <script type="module" src="app.js"> | alongside the page | like defer | yes |

### defer and async only work with src

> On a script written inside the page (no src), defer and async are ignored. Use them for scripts in their own files.

Use defer for your own code. Use async only for scripts that don't depend on the page or on other scripts, like analytics.

## document.readyState

Sometimes your code runs **after** DOMContentLoaded has already fired: a script added later, or one loaded with async. A listener added then waits forever, because the event won't happen again. document.readyState tells you where the page is:

- "loading": the HTML is still being read.
- "interactive": the elements exist; DOMContentLoaded is about to fire or has fired.
- "complete": everything has loaded; load is about to fire or has fired.

This small helper runs your code as soon as the DOM is ready, whether that's now or later:

```
function ready(callback) {
if (document.readyState === "loading") {
document.addEventListener("DOMContentLoaded", callback);
} else {
callback();
}
}

console.log("readyState now:", document.readyState);
ready(() => console.log("The DOM is ready"));
```

### Your script is inside <head> with no defer. What does document.readyState say when it runs?

> - "complete"
> - "interactive"
> - ✓ "loading"

> The browser runs the script as soon as it reaches it, while it is still reading the HTML below it. So the page is still "loading", and nothing in <body> exists yet.

## Common mistakes

### Calling the function instead of passing it

> **Wrong**

```
window.addEventListener("load", start());
```

> **Right**

```
window.addEventListener("load", start);
```

> With brackets, start() runs immediately, and its result (usually undefined) is what gets

handed to addEventListener. Without them, you hand over the function itself, and the browser calls it later.

### Listening for load on document

**Wrong**

```
document.addEventListener("load", () => console.log("never printed"));
```

**Right**

```
window.addEventListener("load", () => console.log("printed"));
```

The page's load event fires on window, not on document. DOMContentLoaded is the other way round: listen for it on document.

### Waiting for load when the DOM is enough

If your click handlers only start after load, a page with large images stays dead for seconds. Wait for DOMContentLoaded (or use defer), not for the pictures.

## Exercise

### When is the page ready?

Make a page with a button and a large image, and a script in <head>. Before you run it, **write down the order** you expect the messages in. Then:

1. Print document.readyState when the script starts.
2. On DOMContentLoaded, print "DOM ready" and the readyState.
3. On load, print "All loaded", the readyState and the image's naturalWidth.
4. Make the button print "Hello!" when it's clicked.

### Answer

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>When is the page ready?</title>
<script>
console.log("1. Script starts:", document.readyState);

document.addEventListener("DOMContentLoaded", () => {
console.log("2. DOM ready:", document.readyState);
const button = document.getElementById("hello");
button.addEventListener("click", () => console.log("Hello!"));
});

window.addEventListener("load", () => {
const photo = document.getElementById("photo");
```

```
console.log("3. All loaded:", document.readyState);
console.log(" Image width:", photo.naturalWidth);
});
</script>
</head>
<body>
<button id="hello">Say hello</button>
<img id="photo" src="https://picsum.photos/2000/1200" alt="" width="300" />
</body>
</html>
```

The order is 1 (loading), 2 (interactive), then 3 (complete, width 2000). The button works because its click handler is added at DOMContentLoaded, when the button exists. Moving the script to its own file with defer would work too, without the DOMContentLoaded wrapper.

Press **Edit** on any example in this lesson to change it and run your own version.

## Key points

- **DOMContentLoaded** (on document): the elements exist. Most code should start here.
- **load** (on window): everything, images included, has loaded.
- Prefer addEventListener to window.onload, which keeps only one function.
- Put your script in <head> with defer, or at the end of <body>.
- document.readyState tells you if the moment has already passed.