Window: setTimeout() method

- Syntax
  - Parameters
  - Return value
  - Exceptions
- Description
  - Working with asynchronous functions
  - Functions are called with the global this
  - Non-number delay values are silently coerced into numbers
  - Maximum delay value
  - Reasons for longer delays than specified
    - Nested timeouts
    - Timeouts in inactive tabs
    - Throttling of tracking scripts
    - Late timeouts
    - Deferral of timeouts during pageload
  - WebExtension background pages and timers
  - Security considerations
- Examples
  - Setting and clearing timeouts
    - HTML
    - JavaScript
    - Result
- Specifications
- Browser compatibility
- See also

> **WARNING:** When the `code` parameter is used, this method dynamically executes its value as JavaScript. APIs like this are known as injection sinks, and are potentially a vector for cross-site-scripting (XSS) attacks.
>
> You can mitigate this risk by always assigning TrustedScript objects instead of strings and enforcing trusted types. See Security considerations for more information.

The **`setTimeout()`** method of the Window interface sets a timer which executes a function or specified piece of code once the timer expires.

Syntax

```
setTimeout(code)
setTimeout(code, delay)

setTimeout(func)
setTimeout(func, delay)
setTimeout(func, delay, param1)
setTimeout(func, delay, param1, param2)
setTimeout(func, delay, param1, param2, /* …, */ paramN)
```

Parameters

- `func`
  - : A Function to be executed after the timer expires.
- `code`
  - : A TrustedScript or a string of arbitrary code that is compiled and executed after `delay` milliseconds. This can be used instead of passing a function, but is *strongly discouraged* for the same reasons that make using eval() a security risk.
- `delay`
  - : A non-negative integer indicating how long the timer should wait before the specified function or code is executed, in milliseconds. Defaults to 0 if not specified. Note:
    - The delay has a maximum value of 2147483647 ms — specifying larger values may result in overflow or a 0 value being used. See maximum delay value

below for more information.
    - The actual delay may be longer than set. For example, setting `delay` to 0 will execute in the next event cycle rather than "immediately". See Reasons for longer delays than specified for more information.
    - If the value isn't a number, implicit type coercion is silently done on the value to convert it to a number. This can lead to unexpected and surprising results — see Non-number delay values are silently coerced into numbers for an example.
    - Negative values behave the same as 0.
- `param1`, ..., `paramN`
    - : Additional arguments which are passed through to the function specified by `func`.

Return value

A positive integer (typically within the range of 1 to 2,147,483,647) that uniquely identifies the timer created by the call. This identifier, often referred to as a "timeout ID", can be passed to clearTimeout() to cancel the timer.

Within the same global environment (e.g., a specific window or worker) the timeout ID is guaranteed not to be reused for any new timer as long as the original timer remains active. However, separate global environments maintain their own independent pools of timer IDs.

Exceptions

- SyntaxError
    - : The `code` can't be parsed as a script.
- TypeError
    - : Thrown if the `code` parameter is set to a string when Trusted Types are enforced by a CSP and no default policy is defined. It is also thrown if the first parameter is not one of the supported types: a function, string or `TrustedScript`.

Description

The `setTimeout()` function is commonly used to call a function that is executed just once, after a delay. You can call Window.clearTimeout() to cancel the timeout before it completes.

If you wish to call a function repeatedly (e.g., every *N* milliseconds), you can use setInterval().

Working with asynchronous functions

`setTimeout()` is an asynchronous function, meaning that it returns immediately after scheduling the callback function or code to run. It does not "wait", blocking execution of the lines of code after `setTimeout()` until the scheduled code has run.

Consider the following example:

```
setTimeout(() => {
  console.log("this is the first message");
}, 5000);
setTimeout(() => {
  console.log("this is the second message");
}, 3000);
setTimeout(() => {
  console.log("this is the third message");
}, 1000);

// Output:

// this is the third message
// this is the second message
// this is the first message
```

The `setTimeout()` method is called three times, passing a callback function that logs the order in which `setTimeout()` was called. Because the earlier methods calls have larger delays, the callback methods are executed in reverse order to which they were scheduled. If `setTimeout()` blocked until the callback completed, the output would display the messages in order.

Asynchronous methods are useful because they allow tasks to be run in parallel when the order of execution does not matter. If the order that an asynchronous method completes does matter, then you can use Promises (promise chaining) to wait on the completion of a task.

Functions are called with the global `this`

The functions passed to `setTimeout()` is run with normal function call semantics for determining the reference of `this`. This problem is explained in detail in the JavaScript reference.

For non-arrow functions, the `this` context is set to the `globalThis` (an alias for `window` in browsers) object.

The following example demonstrates how this can cause unexpected behavior. Here, when we pass the method `counter.count` directly to `setTimeout()`, the `this` context is lost, and the method is called on the global object instead of the `Counter` instance, resulting in a `TypeError` when the `count` method tries to access `this`:

```
class Counter {
  constructor() {
    this.data = new Map();
  }

  count(item) {
    this.data.set(item, (this.data.get(item) || 0) + 1);
  }
}

const counter = new Counter();

counter.count("foo"); // Successfully adds "foo" to the map
setTimeout(counter.count, 1000, "bar");
// TypeError: Cannot read properties of undefined (reading 'set')
```

To work around this, you must make sure that the function passed to `setTimeout` has the correct `this` context. There are three main ways to do this:

1. If you want to explicitly specify the `this` context, instead of passing the method directly, wrap the method call in another anonymous function that explicitly calls the method with the correct context:

   ```
   setTimeout(() => counter.count("bar"), 1000);
   setTimeout(function () {
     counter.count("bar");
   }, 1000);
   ```

2. If you want to use the `this` context of the code that calls `setTimeout()`, always use an arrow function, which inherits the `this` context of its enclosing scope:

   ```
   class Counter {
     // …
     delayedCount(item) {
       // BAD: the `this` context is lost in the callback
       setTimeout(function () {
   ```

```
      this.data.set(item, (this.data.get(item) || 0) + 1);
    }, 1000);
  }
}
```

```
class Counter {
  // …
  delayedCount(item) {
    // GOOD: the arrow function inherits the `this` context of `delayedCount()`
    setTimeout(() => {
      this.data.set(item, (this.data.get(item) || 0) + 1);
    }, 1000);
  }
}
```

3. If you want to avoid extra function wrappers (which increase memory usage) while explicitly specifying the `this` context, you can use the `Function.prototype.bind()` method to create a new function with the correct `this` context:

```
setTimeout(counter.count.bind(counter), 1000, "bar");
```

Non-number delay values are silently coerced into numbers

If `setTimeout()` is called with *delay* value that's not a number, implicit type coercion is silently done on the value to convert it to a number. For example, the following code incorrectly uses the string `"1000"` for the *delay* value, rather than the number `1000` – but it nevertheless works, because when the code runs, the string is coerced into the number `1000`, and so the code executes 1 second later.

```
setTimeout(() => {
  console.log("Delayed for 1 second.");
}, "1000");
```

In many cases, the implicit type coercion can lead to unexpected and surprising results. For example, when the following code runs, the string `"1 second"` ultimately gets coerced into the number `0` — and so, the code executes with zero delay.

```
setTimeout(() => {
  console.log("Delayed for 1 second.");
}, "1 second");
```

Therefore, don't use strings for the *delay* value but instead always use numbers:

```
setTimeout(() => {
  console.log("Delayed for 1 second.");
}, 1000);
```

Maximum delay value

The `delay` argument is converted to a signed 32-bit integer, which limits the value to 2147483647 ms, or roughly 24.8 days. Delays of more than this value will cause an integer overflow. So for example, this code:

```
setTimeout(() => console.log("hi!"), 2 ** 32 - 5000);
```

...results in the timeout being executed immediately (since `2**32 - 5000` overflows to a negative number), while the following code:

```
setTimeout(() => console.log("hi!"), 2 ** 32 + 5000);
```

...results in the timeout being executed after approximately 5 seconds.

> **NOTE:** In Node.js, any timeout larger than 2,147,483,647 ms results in immediate execution.

Reasons for longer delays than specified

There are a number of reasons why a timeout may take longer to fire than anticipated. This section describes the most common reasons.

Nested timeouts

As specified in the HTML standard, browsers will enforce a minimum timeout of 4 milliseconds once a nested call to `setTimeout` has been scheduled 5 times.

This can be seen in the following example, in which we nest a call to `setTimeout` with a delay of `0` milliseconds, and log the delay each time the handler is called. The first four times, the delay is approximately 0 milliseconds, and after that it is approximately 4 milliseconds:

```
<button id="run">Run</button>
<table>
  <thead>
    <tr>
      <th>Previous</th>
      <th>This</th>
      <th>Actual delay</th>
    </tr>
  </thead>
  <tbody id="log"></tbody>
</table>
```

```
let last = 0;
let iterations = 10;

function timeout() {
  // log the time of this call
  log(new Date().getMilliseconds());
  // if we are not finished, schedule the next call
  if (iterations-- > 0) {
    setTimeout(timeout, 0);
  }
}

function run() {
  // clear the log
  const log = document.querySelector("#log");
  while (log.lastElementChild) {
    log.removeChild(log.lastElementChild);
  }

  // initialize iteration count and the starting timestamp
  iterations = 10;
  last = new Date().getMilliseconds();
  // start timer
  setTimeout(timeout, 0);
```

```
}

function log(now) {
  // log the last timestamp, the new timestamp, and the difference
  const tableBody = document.getElementById("log");
  const logRow = tableBody.insertRow();
  logRow.insertCell().textContent = last;
  logRow.insertCell().textContent = now;
  logRow.insertCell().textContent = now - last;
  last = now;
}

document.querySelector("#run").addEventListener("click", run);
```

```
* {
  font-family: monospace;
}
th,
td {
  padding: 0 10px;
  text-align: center;
  border: 1px solid;
}
table {
  border-collapse: collapse;
  margin-top: 10px;
}
```

Timeouts in inactive tabs

To reduce the load (and associated battery usage) from background tabs, browsers will enforce a minimum timeout delay in inactive tabs. It may also be waived if a page is playing sound using a Web Audio API AudioContext.

The specifics of this are browser-dependent:

- Firefox Desktop has a minimum timeout of 1 second for inactive tabs.
- Firefox for Android has a minimum timeout of 15 minutes for inactive tabs and may unload them entirely.
- Firefox does not throttle inactive tabs if the tab contains an AudioContext.
- Chrome uses different levels of throttling depending on the tab activity:

- **Minimal throttling**: Applies to timers when the page is visible, has made sound recently, or is otherwise considered active by Chrome. Timers run close to the requested interval.
- **Throttling**: Applies to timers when minimal throttle conditions are not met and any of these conditions are true:
  - Nesting count (i.e., number of chained timer calls) is lower than 5.
  - Page has been invisible for less than 5 minutes.
  - WebRTC is active.

  Timers in this state are checked once per second, which may be batched together with other timers that have similar timeouts.
- **Intensive throttling**: Introduced in Chrome 88 (January 2021). Applies to timers when neither minimal throttling nor throttling conditions are met, and all of the following conditions are met:
  - Nesting count is 5 or higher.
  - Page has been invisible for more than 5 minutes.
  - Page has been silent for more than 30 seconds.
  - WebRTC is inactive.

  Timers in this state are checked once per minute, which may be batched together with other timers that have similar timeouts.

Throttling of tracking scripts

Firefox enforces additional throttling for scripts that it recognizes as tracking scripts. When running in the foreground, the throttling minimum delay is still 4ms. In background tabs, however, the throttling minimum delay is 10,000 ms, or 10 seconds, which comes into effect 30 seconds after a document has first loaded.

See Tracking Protection for more details.

Late timeouts

The timeout can also fire later than expected if the page (or the OS/browser) is busy with other tasks. One important case to note is that the function or code snippet cannot be executed until the thread that called `setTimeout()` has terminated. For example:

```
function foo() {
  console.log("foo has been called");
}
setTimeout(foo, 0);
console.log("After setTimeout");
```

Will write to the console:

```
After setTimeout
foo has been called
```

This is because even though `setTimeout` was called with a delay of zero, it's placed on a queue and scheduled to run at the next opportunity; not immediately. Currently-executing code must complete before functions on the queue are executed, thus the resulting execution order may not be as expected.

Deferral of timeouts during pageload

Firefox will defer firing `setTimeout()` timers while the current tab is loading. Firing is deferred until the main thread is deemed idle (similar to Window.requestIdleCallback()), or until the load event is fired.

WebExtension background pages and timers

In WebExtensions, `setTimeout()` does not work reliably. Extension authors should use the `alarms` API instead.

Security considerations

The method can be used to execute arbitrary input passed in the `code` parameter. If the input is a potentially unsafe string provided by a user, this is a possible vector for Cross-site-scripting (XSS) attacks.

For example, the following code shows how `setTimeout()` might execute `untrustedCode` provided by a user:

```
const untrustedCode = "alert('Potentially evil code!');";
const id = setTimeout(untrustedCode, 1000);
```

Websites with a Content Security Policy (CSP) that specifies `script-src` or `default-src` will prevent such code running by default. You can specify `unsafe-eval` in your CSP to allow `setTimeout()` to execute, but this is unsafe as it disables one of the main protections of CSP. See Inline JavaScript in the CSP guide.

If you must allow the scripts to run via `setTimeout()` you can mitigate these issues by always assigning TrustedScript objects instead of strings, and enforcing trusted types using the `require-`

`trusted-types-for` CSP directive. This ensures that the input is passed through a transformation function.

To allow `setTimeout()` to run, you will additionally need to specify the `trusted-types-eval` keyword in your CSP `script-src` directive. This acts in the same way as `unsafe-eval`, but *only* allows the method to evaluate if trusted types are enabled (if you were to use `unsafe-eval` it would allow execution even on browsers that do not support trusted types).

For example, the required CSP for your site might look like this:

```
Content-Security-Policy: require-trusted-types-for 'script'; script-src '<your_allowlist>'
'trusted-types-eval'
```

The behavior of the transformation function will depend on the specific use case that requires a user-provided script. If possible you should lock the allowed scripts to exactly the code that you trust to run. If that is not possible, you might allow or block the use of certain functions within the provided string.

Examples

Note that these examples omit the use of trusted types for brevity. See Using `TrustedScript` in `eval()` for code showing the expected approach.

Setting and clearing timeouts

The following example sets up two simple buttons in a web page and hooks them to the `setTimeout()` and `clearTimeout()` routines. Pressing the first button will set a timeout which shows a message after two seconds and stores the timeout id for use by `clearTimeout()`. You may optionally cancel this timeout by pressing on the second button.

HTML

```
<button id="show">Show a message after two seconds</button>
<button id="cancel">Cancel message before it happens</button>

<div id="output"></div>
```

JavaScript

```
let timeoutID;

function setOutput(outputContent) {
  document.querySelector("#output").textContent = outputContent;
}

function delayedMessage() {
  setOutput("");
  timeoutID = setTimeout(setOutput, 2 * 1000, "That was really slow!");
}

function clearMessage() {
  clearTimeout(timeoutID);
}

document.getElementById("show").addEventListener("click", delayedMessage);
document.getElementById("cancel").addEventListener("click", clearMessage);
```

```
#output {
  padding: 0.5rem 0;
}
```

Result

See also the clearTimeout() example.

Specifications

Browser compatibility

See also

- Polyfill of `setTimeout` which allows passing arguments to the callback in `core-js`
- Window.clearTimeout()
- WorkerGlobalScope.setTimeout()

- Window.setInterval()
- Window.requestAnimationFrame()
- Window.queueMicrotask()

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:35:21 UTC