MDN Web Docs
moz://a

setTimeout()

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

Syntax

```
var timeoutID = setTimeout(function[, delay, arg1, arg2, ...]);
var timeoutID = setTimeout(function[, delay]);
var timeoutID = setTimeout(code[, delay]);
```

Parameters

`function`

A `function` to be executed after the timer expires.

`code`

An alternative syntax that allows you to include a string instead of a function, which is compiled and executed when the timer expires. This syntax is **not recommended** for the same reasons that make using `eval()` a security risk.

`delay` Optional

The time, in milliseconds that the timer should wait before the specified function or code is executed. If this parameter is omitted, a value of 0 is used, meaning execute "immediately", or more accurately, the next event cycle. Note that in either case, the actual delay may be longer than intended; see Reasons for delays longer than specified below.

`arg1, ..., argN` Optional

Additional arguments which are passed through to the function specified by `function`.

Return value

The returned `timeoutID` is a positive integer value which identifies the timer created by the call

to `setTimeout()`. This value can be passed to `clearTimeout()` to cancel the timeout.

It is guaranteed that a `timeoutID` value will never be reused by a subsequent call to `setTimeout()` or `setInterval()` on the same object (a window or a worker). However,

different objects use separate pools of IDs.

Description

Timeouts are cancelled using `clearTimeout()`.

To call a function repeatedly (e.g., every *N* milliseconds), consider using `setInterval()`.

Working with asynchronous functions

`setTimeout()` is an asynchronous function, meaning that the timer function will not pause execution of other functions in the functions stack. In other words, you cannot use `setTimeout()` to create a "pause" before the next function in the function stack fires.

See 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
```

Notice that the first function does not create a 5-second "pause" before calling the second function. Instead, the first function is called, but waits 5 seconds to execute. While the first function is waiting to execute, the second function is called, and a 3-second wait is applied to the second function before it executes. Since neither the first nor the second function's timers have completed, the third function is called and completes its execution first. Then the second follows. Then finally the first function is executed after its timer finally completes.

To create a progression in which one function only fires after the completion of another function,

see the documentation on Promises.

The "this" problem

When you pass a method to `setTimeout()`, it will be invoked with a `this` value that may differ from your expectation. The general issue is explained in detail in the JavaScript reference.

Code executed by `setTimeout()` is called from an execution context separate from the function from which `setTimeout` was called. The usual rules for setting the `this` keyword for the called function apply, and if you have not set `this` in the call or with `bind`, it will default to the `window` (or `global`) object. It will not be the same as the `this` value for the function that called `setTimeout`.

See the following example:

```
const myArray = ['zero', 'one', 'two'];
myArray.myMethod = function (sProperty) {
  console.log(arguments.length > 0 ? this[sProperty] : this);
};

myArray.myMethod(); // prints "zero,one,two"
myArray.myMethod(1); // prints "one"
```

The above works because when `myMethod` is called, its `this` is set to `myArray` by the call, so within the function, `this[sProperty]` is equivalent to `myArray[sProperty]`. However, in the following:

```
setTimeout(myArray.myMethod, 1.0*1000); // prints "[object Window]" after
setTimeout(myArray.myMethod, 1.5*1000, '1'); // prints "undefined" after 1
```

The `myArray.myMethod` function is passed to `setTimeout`, then when it's called, its `this` is not set so it defaults to the `window` object.

There's also no option to pass a `thisArg` to `setTimeout` as there is in Array methods such as `forEach()` and `reduce()`. As shown below, using `call` to set `this` doesn't work either.

```
setTimeout.call(myArray, myArray.myMethod, 2.0*1000); // error
```

```
setTimeout.call(myArray, myArray.myMethod, 2.5*1000, 2); // same error
```

Solutions

Use a wrapper function

A common way to solve the problem is to use a wrapper function that sets `this` to the required value:

```
setTimeout(function(){myArray.myMethod()}, 2.0*1000); // prints "zero,one,
setTimeout(function(){myArray.myMethod('1')}, 2.5*1000); // prints "one" a
```

The wrapper function can be an arrow function:

```
setTimeout(() => {myArray.myMethod()}, 2.0*1000); // prints "zero,one,two"
setTimeout(() => {myArray.myMethod('1')}, 2.5*1000); // prints "one" after
```

Use bind()

Alternatively, you can use [bind()](#) to set the value of `this` for all calls to a given function:

```
const myArray = ['zero', 'one', 'two'];
const myBoundMethod = (function (sProperty) {
    console.log(arguments.length > 0 ? this[sProperty] : this);
}).bind(myArray);

myBoundMethod(); // prints "zero,one,two" because 'this' is bound to myArray i
myBoundMethod(1); // prints "one"
setTimeout(myBoundMethod, 1.0*1000); // still prints "zero,one,two" after 1 se
setTimeout(myBoundMethod, 1.5*1000, "1"); // prints "one" after 1.5 seconds
```

Passing string literals

Passing a string instead of a function to `setTimeout()` has the same problems as using [eval()](#).

```
// Don't do this
setTimeout("console.log('Hello World!');", 500);
```

```
// Do this instead
setTimeout(function() {
  console.log('Hello World!');
}, 500);
```

A string passed to **`setTimeout()`** is evaluated in the global context, so local symbols in the context where **`setTimeout()`** was called will not be available when the string is evaluated as code.

Reasons for delays longer 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>
<pre>previous    this    actual delay</pre>
<div id="log"></div>
```

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

function timeout() {
  // log the time of this call
  logline(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 pad(number) {
  return number.toString().padStart(3, "0");
}

function logline(now) {
  // log the last timestamp, the new timestamp, and the difference
  const newLine = document.createElement("pre");
  newLine.textContent = `${pad(last)}         ${pad(now)}          ${now - las
  document.getElementById("log").appendChild(newLine);
  last = now;
}

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

Run

```
previous    this    actual delay
615         616          1
616         618          2
618         619          1
619         622          3
622         631          9
631         636          5
636         642          6
642         647          5
647         655          8
655         659          4
659         665          6
```

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 and Chrome both have 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`.

Throttling of tracking scripts

Firefox enforces additional throttling for scripts that it recognises 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

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.

Maximum delay value

Browsers including Internet Explorer, Chrome, Safari, and Firefox store the delay as a 32-bit

signed integer internally. This causes an integer overflow when using delays larger than 2,147,483,647 ms (about 24.8 days), resulting in the timeout being executed immediately.

Examples

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 onclick="delayedMessage();">Show an message after two seconds</but
<button onclick="clearMessage();">Cancel message before it happens</buttor

<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);
}
```

Result