**[Event Loop](https://javascript.info/event-loop" \l "event-loop)**

The *event loop* concept is very simple. There’s an endless loop, where the JavaScript engine waits for tasks, executes them and then sleeps, waiting for more tasks.

The general algorithm of the engine:

1. While there are tasks:
   * execute them, starting with the oldest task.
2. Sleep until a task appears, then go to 1.

That’s a formalization for what we see when browsing a page. The JavaScript engine does nothing most of the time, it only runs if a script/handler/event activates.

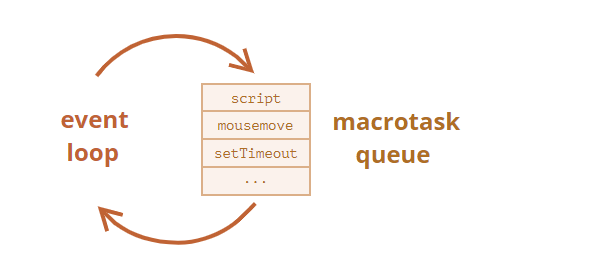
Examples of tasks:

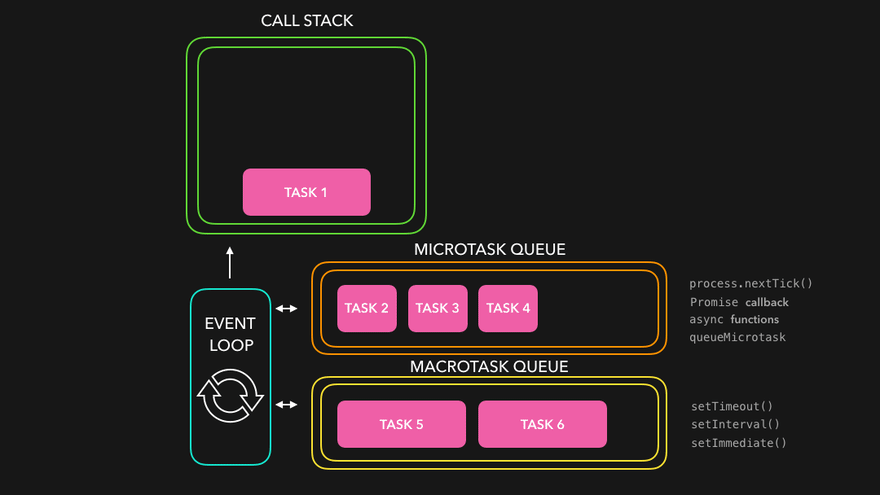
* When an external script <script src="..."> loads, the task is to execute it.
* When a user moves their mouse, the task is to dispatch mousemove event and execute handlers.
* When the time is due for a scheduled setTimeout, the task is to run its callback.
* …and so on.

Tasks are set – the engine handles them – then waits for more tasks (while sleeping and consuming close to zero CPU).

It may happen that a task comes while the engine is busy, then it’s enqueued.

The tasks form a queue, so-called “macrotask queue” (v8 term):





For instance, while the engine is busy executing a script, a user may move their mouse causing mousemove, and setTimeout may be due and so on, these tasks form a queue, as illustrated on the picture above.

Tasks from the queue are processed on “first come – first served” basis. When the engine browser is done with the script, it handles mousemove event, then setTimeout handler, and so on.

So far, quite simple, right?

Two more details:

1. Rendering never happens while the engine executes a task. It doesn’t matter if the task takes a long time. Changes to the DOM are painted only after the task is complete.
2. If a task takes too long, the browser can’t do other tasks, such as processing user events. So after a time, it raises an alert like “Page Unresponsive”, suggesting killing the task with the whole page. That happens when there are a lot of complex calculations or a programming error leading to an infinite loop.

[**Macrotasks and Microtasks**](https://javascript.info/event-loop#macrotasks-and-microtasks)

Along with *macrotasks*, described in this chapter, there are *microtasks*, mentioned in the chapter [Microtasks](https://javascript.info/microtask-queue).

Microtasks come solely from our code. They are usually created by promises: an execution of .then/catch/finally handler becomes a microtask. Microtasks are used “under the cover” of await as well, as it’s another form of promise handling.

There’s also a special function queueMicrotask(func) that queues func for execution in the microtask queue.

**Immediately after every *macrotask*, the engine executes all tasks from *microtask* queue, prior to running any other macrotasks or rendering or anything else.**

For instance, take a look:

setTimeout(() => alert("timeout"));

Promise.resolve()

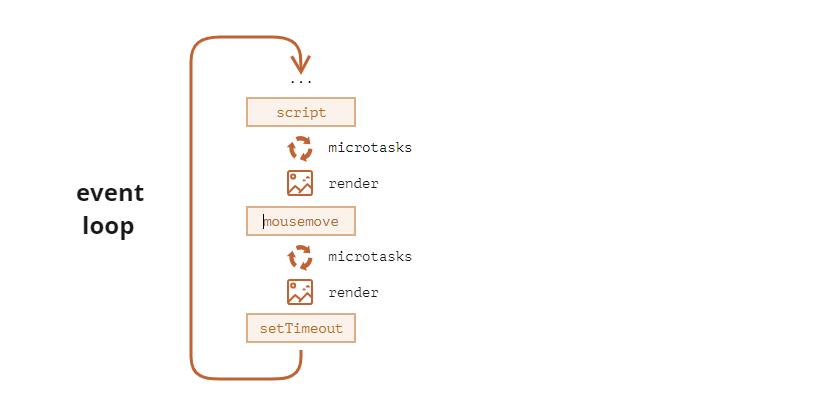
.then(() => alert("promise"));

alert("code");

What’s going to be the order here?

1. code shows first, because it’s a regular synchronous call.
2. promise shows second, because .then passes through the microtask queue, and runs after the current code.
3. timeout shows last, because it’s a macrotask.

The richer event loop picture looks like this (order is from top to bottom, that is: the script first, then microtasks, rendering and so on):



All microtasks are completed before any other event handling or rendering or any other macrotask takes place.

That’s important, as it guarantees that the application environment is basically the same (no mouse coordinate changes, no new network data, etc) between microtasks.

If we’d like to execute a function asynchronously (after the current code), but before changes are rendered or new events handled, we can schedule it with queueMicrotask.

Here’s an example with “counting progress bar”, similar to the one shown previously, but queueMicrotask is used instead of setTimeout. You can see that it renders at the very end. Just like the synchronous code:

<div id="progress"></div>

<script>

let i = 0;

function count() {

// do a piece of the heavy job (\*)

do {

i++;

progress.innerHTML = i;

} while (i % 1e3 != 0);

if (i < 1e6) {

queueMicrotask(count);

}

}

count();

</script>

## [Summary](https://javascript.info/event-loop" \l "summary)

A more detailed event loop algorithm (though still simplified compared to the [specification](https://html.spec.whatwg.org/multipage/webappapis.html#event-loop-processing-model)):

1. Dequeue and run the oldest task from the macrotask queue (e.g. “script”).
2. Execute all microtasks:
   * While the microtask queue is not empty:
     + Dequeue and run the oldest microtask.
3. Render changes if any.
4. If the macrotask queue is empty, wait till a macrotask appears.
5. Go to step 1.

To schedule a new macrotask:

* Use zero delayed setTimeout(f).

That may be used to split a big calculation-heavy task into pieces, for the browser to be able to react to user events and show progress between them.

Also, used in event handlers to schedule an action after the event is fully handled (bubbling done).

To schedule a new microtask

* Use queueMicrotask(f).
* Also promise handlers go through the microtask queue.

There’s no UI or network event handling between microtasks: they run immediately one after another.

So one may want to queueMicrotask to execute a function asynchronously, but within the environment state.

**Web Workers**

For long heavy calculations that shouldn’t block the event loop, we can use [Web Workers](https://html.spec.whatwg.org/multipage/workers.html).

That’s a way to run code in another, parallel thread.

Web Workers can exchange messages with the main process, but they have their own variables, and their own event loop.

Web Workers do not have access to DOM, so they are useful, mainly, for calculations, to use multiple CPU cores simultaneously.

## [Tasks](https://javascript.info/event-loop#tasks)

### [What will be the output of this code?](https://javascript.info/event-loop" \l "what-will-be-the-output-of-this-code)

importance: 5

console.log(1);

setTimeout(() => console.log(2));

Promise.resolve().then(() => console.log(3));

Promise.resolve().then(() => setTimeout(() => console.log(4)));

Promise.resolve().then(() => console.log(5));

setTimeout(() => console.log(6));

console.log(7);

solution

The console output is: 1 7 3 5 2 6 4.

The task is quite simple, we just need to know how microtask and macrotask queues work.

Let’s see what’s going on, step by step.

console.log(1);

// The first line executes immediately, it outputs `1`.

// Macrotask and microtask queues are empty, as of now.

setTimeout(() => console.log(2));

// `setTimeout` appends the callback to the macrotask queue.

// - macrotask queue content:

// `console.log(2)`

Promise.resolve().then(() => console.log(3));

// The callback is appended to the microtask queue.

// - microtask queue content:

// `console.log(3)`

Promise.resolve().then(() => setTimeout(() => console.log(4)));

// The callback with `setTimeout(...4)` is appended to microtasks

// - microtask queue content:

// `console.log(3); setTimeout(...4)`

Promise.resolve().then(() => console.log(5));

// The callback is appended to the microtask queue

// - microtask queue content:

// `console.log(3); setTimeout(...4); console.log(5)`

setTimeout(() => console.log(6));

// `setTimeout` appends the callback to macrotasks

// - macrotask queue content:

// `console.log(2); console.log(6)`

console.log(7);

// Outputs 7 immediately.

To summarize,

1. Numbers 1 and 7 show up immediately, because simple console.log calls don’t use any queues.
2. Then, after the main code flow is finished, the microtask queue runs.
   * It has commands: console.log(3); setTimeout(...4); console.log(5).
   * Numbers 3 and 5 show up, while setTimeout(() => console.log(4)) adds the console.log(4) call to the end of the macrotask queue.
   * The macrotask queue is now: console.log(2); console.log(6); console.log(4).
3. After the microtask queue becomes empty, the macrotask queue executes. It outputs 2, 6, 4.

Finally, we have the output: 1 7 3 5 2 6 4.

**Event Loop:** An Event Loop in JavaScript is said to be a constantly running process that keeps a tab on the call stack. Its main function is to check whether the call stack is empty or not. If the call stack turns out to be empty, the event loop proceeds to execute all the callbacks waiting in the task queue. Inside the task queue, the tasks are broadly classified into two categories, namely micro-tasks and macro-tasks.

**Micro-tasks within an event loop:** A micro-task is said to be a function that is executed after the function or program which created it exits and only if the JavaScript execution stack is empty, but before returning control to the event loop being used by the user agent to drive the script’s execution environment. A Micro-task is also capable of en-queuing other micro-tasks.

Micro-tasks are often scheduled for things that are required to be completed immediately after the execution of the current script. On completion of one macro-task, the event loop moves on to the micro-task queue. The event loop does not move to the next task outside of the micro-task queue until all the tasks inside the micro-task queue are completed. This implies that the micro-task queue has a higher priority.

Once all the tasks inside the micro-task queue are finished, only then does the event loop shifts back to the macro-task queue. The primary reason for prioritizing the micro-task queue is to improve the user experience. The micro-task queue is processed after callbacks given that any other JavaScript is not under mid-execution. Micro-tasks include mutation observer callbacks as well as promise callbacks.

In such a case wherein new micro-tasks are being added to the queue, these additional micro-tasks are added at the end of the micro-queue and these are also processed. This is because the event loop will keep on calling micro-tasks until there are no more micro-tasks left in the queue, even if new tasks keep getting added. Another important reason for using micro-tasks is to ensure consistent ordering of tasks as well as simultaneously reducing the risk of delays caused by users.

**Syntax:**Adding micro-tasks:

queueMicrotask(() => {

// Code to be run inside the micro-task

});

The micro-task function itself takes no parameters and does not return a value.

**Examples:** process.nextTick, Promises, queueMicrotask, MutationObserver

**Macro-tasks within an event loop:** Macro-task represents some discrete and independent work. These are always the execution of the JavaScript code and the micro-task queue is empty. Macro-task queue is often considered the same as the task queue or the event queue. However, the macro-task queue works the same as the task queue. The only small difference between the two is that the task queue is used for synchronous statements whereas the macro-task queue is used for asynchronous statements.

In JavaScript, no code is allowed to execute until an event has occurred. {It is worth mentioning that the execution of a JavaScript code execution is itself a macro-task.} The event is queued as a macro-task. When a (macro) task, present in the macro-task queue is being executed, new events may be registered and in turn, created and added to the queue.

Up on initialization, the JavaScript engine first pulls off the first task in the macro-task queue and executes the callback handler. The JavaScript engine then sends these asynchronous functions to the API module, and the module pushes them to the macro-task queue at the right time. Once inside the macro-task queue, each macro-task is required to wait for the next round of the event loop. In this way, the code is executed.

All micro-tasks logged are processed in one fell swoop in a single macro-task execution cycle. In comparison, the macro-task queue has a lower priority. Macro tasks include parsing HTML, generating DOM, executing main thread JavaScript code, and other events such as page loading, input, network events, timer events, etc.

**Examples:**setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI Rendering