mdn web docs

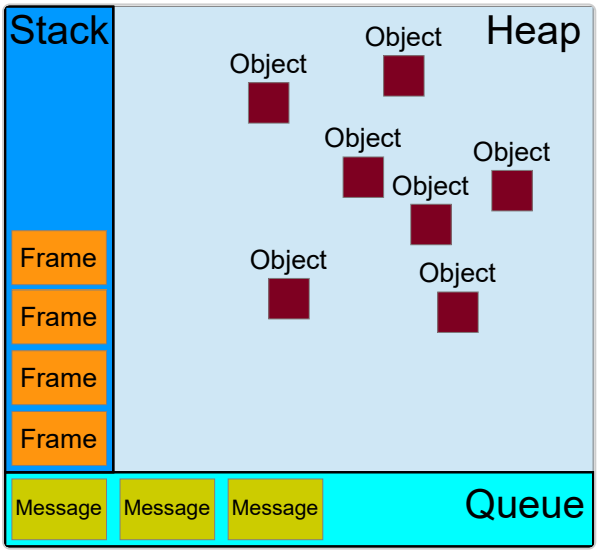
The event loop

JavaScript has a runtime model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.

Runtime concepts

The following sections explain a theoretical model. Modern JavaScript engines implement and heavily optimize the described semantics.

Visual representation



Stack

Function calls form a stack of *frames*.

JS

```
function foo(b) {
  const a = 10;
  return a + b + 11;
```

```
}

function bar(x) {
  const y = 3;
  return foo(x * y);
}

const baz = bar(7); // assigns 42 to baz
```

Order of operations:

1. When calling `bar`, a first frame is created containing references to `bar`'s arguments and local variables.
2. When `bar` calls `foo`, a second frame is created and pushed on top of the first one, containing references to `foo`'s arguments and local variables.
3. When `foo` returns, the top frame element is popped out of the stack (leaving only `bar`'s call frame).
4. When `bar` returns, the stack is empty.

Note that the arguments and local variables may continue to exist, as they are stored outside the stack — so they can be accessed by any nested functions long after their outer function has returned.

Heap

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

Queue

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function that gets called to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input

parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

Event loop

The **event loop** got its name because of how it's usually implemented, which usually resembles:

JS

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

`queue.waitForMessage()` waits synchronously for a message to arrive (if one is not already available and waiting to be handled).

"Run-to-completion"

Each message is processed completely before any other message is processed.

This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be preempted and will run entirely before any other code runs (and can modify data the function manipulates). This differs from C, for instance, where if a function runs in a thread, it may be stopped at any point by the runtime system to run some other code in another thread.

A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog. A good practice to follow is to make message processing short and if possible cut down one message into several messages.

Adding messages

In web browsers, messages are often added when an event occurs and there is an event listener attached to it. If there is no listener, the event is lost. So a click on an element with a click event handler will add a message — likewise with any other event. However, some events happen synchronously without a message — for example, simulated clicks via the `click` method.

The first two arguments to the function `setTimeout()` are a message to add to the queue and a time value (optional; defaults to `0`). The *time value* represents the (minimum) delay after which the message will be pushed into the queue. If there is no other message in the queue, and the stack is empty, the message is processed right after the delay. However, if there are messages, the `setTimeout()` message will have to wait for other messages to be processed. For this reason, the second argument indicates a *minimum* time — not a *guaranteed* time.

Here is an example that demonstrates this concept (`setTimeout()` does not run immediately after its timer expires):

JS

```
const seconds = new Date().getTime() / 1000;

setTimeout(() => {
  // prints out "2", meaning that the callback is not called immediately after 500
milliseconds.
  console.log(`Ran after ${new Date().getTime() / 1000 - seconds} seconds`);
}, 500);

while (true) {
  if (new Date().getTime() / 1000 - seconds >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

Zero delays

Zero delay doesn't mean the call back will fire-off after zero milliseconds. Calling `setTimeout()` with a delay of `0` (zero) milliseconds doesn't execute the callback function after the given interval.

The execution depends on the number of waiting tasks in the queue. In the example below, the message `"this is just a message"` will be written to the console before the message in the callback gets processed, because the delay is the *minimum* time required for the runtime to process the request (not a *guaranteed* time).

The `setTimeout()` needs to wait for all the code for queued messages to complete even though you specified a particular time limit for your `setTimeout()`.

JS

```
(() => {
  console.log("this is the start");

  setTimeout(() => {
    console.log("Callback 1: this is a msg from call back");
  }); // has a default time value of 0

  console.log("this is just a message");

  setTimeout(() => {
    console.log("Callback 2: this is a msg from call back");
  }, 0);

  console.log("this is the end");
})();

// "this is the start"
// "this is just a message"
// "this is the end"
// "Callback 1: this is a msg from call back"
// "Callback 2: this is a msg from call back"
```

Several runtimes communicating together

A web worker or a cross-origin `iframe` has its own stack, heap, and message queue. Two distinct runtimes can only communicate through sending messages via the `postMessage` method. This method adds a message to the other runtime if the latter listens to `message` events.

Never blocking

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks. Handling I/O is typically performed via events and callbacks, so when the application is waiting for an IndexedDB query to return or a `fetch()` request to return, it can still process other things like user input.

Legacy exceptions exist like `alert` or synchronous XHR, but it is considered good practice to avoid them. Beware: exceptions to the exception do exist (but are usually implementation bugs, rather than anything else).

See also

- Event loops in the HTML standard
- What is the Event Loop? in the Node.js docs

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Oct 7, 2024 by MDN contributors.