Using Web Workers

see all contributors

Web Workers provide a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface. In addition, they can perform I/O using `XMLHttpRequest` (although the `responseXML` and `channel` attributes are always null). Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa.) This article provides a detailed introduction to using web workers.

Web Workers API

A worker is an object created using a constructor (e.g. `Worker()`) that runs a named JavaScript file — this file contains the code that will run in the worker thread; workers run in another global context that is different from the current `window`. Thus, using the `window` shortcut to get the current global scope (instead of `self`) within a `Worker` will return an error.

The worker context is represented by a `DedicatedWorkerGlobalScope` object in the case of dedicated workers (standard workers that are utilized by a single script; shared workers use `SharedWorkerGlobalScope`). A dedicated worker is only accessible from the script that first spawned it, whereas shared workers can be accessed from multiple scripts.

Note: See The Web Workers API landing page for reference documentation on workers and additional guides.

You can run whatever code you like inside the worker thread, with some exceptions. For example, you can't directly manipulate the DOM from inside a worker, or use some default methods and properties of the `window` object. But you can use a large number of items available under `window`, including WebSockets, and data storage mechanisms like IndexedDB and the Firefox OS-only Data Store API. See Functions and classes available to workers for more details.

Data is sent between workers and the main thread via a system of messages — both sides send their messages using the `postMessage()` method, and respond to messages via the `onmessage` event handler (the message is contained within the `Message` event's data attribute.) The data is copied rather than shared.

Workers may, in turn, spawn new workers, as long as those workers are hosted within the same origin as the parent page. In addition, workers may use `XMLHttpRequest` for network I/O, with the exception that the `responseXML` and `channel` attributes on `XMLHttpRequest` always return `null`.

Dedicated workers

As mentioned above, a dedicated worker is only accessible by the script that called it. In this section we'll discuss the JavaScript found in our Basic dedicated worker example (run dedicated worker): This allows you to enter two numbers to be multiplied together. The numbers are sent to a dedicated worker, multiplied together, and the result is returned to the page and displayed.

This example is rather trivial, but we decided to keep it simple while introducing you to basic worker concepts. More advanced details are covered later on in the article.

Worker feature detection

For slightly more controlled error handling and backwards compatibility, it is a good idea to wrap your worker accessing code in the following (main.js):

```
if (window.Worker) {

  ...

}
```

Spawning a dedicated worker

Creating a new worker is simple. All you need to do is call the `Worker()` constructor, specifying the URI of a script to execute in the worker thread (main.js):

```
var myWorker = new Worker("worker.js");
```

Sending messages to and from a dedicated worker

The magic of workers happens via the `postMessage()` method and the `onmessage` event handler. When you want to send a message to the worker, you post messages to it like this (main.js):

```
first.onchange = function() {
  myWorker.postMessage([first.value,second.value]);
  console.log('Message posted to worker');
}

second.onchange = function() {
  myWorker.postMessage([first.value,second.value]);
  console.log('Message posted to worker');
}
```

So here we have two `<input>` elements represented by the variables `first` and `second`; when the value of either is changed, `myWorker.postMessage([first.value,second.value])` is used to send the value inside both to the worker, as an array. You can send pretty much anything you like in the message.

In the worker, we can respond when the message is received by writing an event handler block like this (worker.js):

```
onmessage = function(e) {
  console.log('Message received from main script');
  var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
  console.log('Posting message back to main script');
  postMessage(workerResult);
}
```

The `onmessage` handler allows us to run some code whenever a message is received, with the message itself being available in the `message` event's `data` attribute. Here we simply multiply together the two numbers then use `postMessage()` again, to post the result back to the main thread.

Back in the main thread, we use `onmessage` again, to respond to the message sent back from the worker:

```
myWorker.onmessage = function(e) {
  result.textContent = e.data;
  console.log('Message received from worker');
}
```

Here we grab the message event data and set it as the `textContent` of the result paragraph, so the user can see the result of the calculation.

Note: The URI passed as a parameter to the `Worker` constructor must obey the same-origin policy .

There is currently disagreement among browsers vendors on what URIs are of the same-origin; Gecko 10.0 (Firefox 10.0 / Thunderbird 10.0 / SeaMonkey 2.7) and later do allow data URIs and Internet Explorer 10 does not allow Blob URIs as a valid script for workers.

Note: Notice that `onmessage` and `postMessage()` need to be hung off the `Worker` object when used in the main script thread, but not when used in the worker. This is because, inside the worker, the worker is effectively the global scope.

Note: When a message is passed between the main thread and worker, it is copied or "transferred" (moved), not shared. Read Transferring data to and from workers: further details for a much more thorough explanation.

Terminating a worker

If you need to immediately terminate a running worker from the main thread, you can do so by calling the worker's `terminate` method:

```
myWorker.terminate();
```

The worker thread is killed immediately without an opportunity to complete its operations or clean up after itself.

In the worker thread, workers may close themselves by calling their own `close` method:

```
close();
```

Handling errors

When a runtime error occurs in the worker, its `onerror` event handler is called. It receives an event named `error` which implements the `ErrorEvent` interface.

The event doesn't bubble and is cancelable; to prevent the default action from taking place, the worker can call the error event's `preventDefault()` method.

The error event has the following three fields that are of interest:

`message`
A human-readable error message.

`filename`
The name of the script fil in which the error occurred.

`lineno`
The line number of the script file on which the error occurred.

Spawning subworkers

Workers may spawn more workers if they wish. So-called sub-workers must be hosted within the same origin as the parent page. Also, the URIs for subworkers are resolved relative to the parent worker's location rather than that of the owning page. This makes it easier for workers to keep track of where their dependencies are.

Importing scripts and libraries

Worker threads have access to a global function, `importScripts()`, which lets them import scripts in the same domain into their scope. It accepts zero or more URIs as parameters to resources to import; all of the following examples are valid:

```
importScripts();                         /* imports nothing *
importScripts('foo.js');                 /* imports just "foo
importScripts('foo.js', 'bar.js');       /* imports two scrip
```

The browser loads each listed script and executes it. Any global objects from each script may then be used by the worker. If the script can't be loaded, `NETWORK_ERROR` is thrown, and subsequent code will not be executed. Previously executed code (including code deferred using `window.setTimeout()`) will still be functional though. Function declarations **after** the `importScripts()` method are also kept, since these are always evaluated before the rest of the code.

> **Note**: Scripts may be downloaded in any order, but will be executed in the order in which you pass the filenames into `importScripts()` . This is done synchronously; `importScripts()` does not return until all the scripts have been loaded and executed.

Shared workers

A shared worker is accessible by multiple scripts — even if they are being accessed by different windows, iframes or even workers. In this section we'll discuss the JavaScript found in our Basic shared worker example (run shared worker): This is very similar to the basic dedicated worker example, except that it has two functions available handled by different script files: *multiplying two numbers*, or *squaring a number*. Both scripts use the same worker to do the actual calculation required.

Here we'll concentrate on the differences between dedicated and shared workers. Note that in this example we have two HTML pages, each with JavaScript applied that uses the same single worker file.

> **Note**: If SharedWorker can be accessed from several browsing contexts, all those browsing contexts must share the exact same origin (same protocol, host, and port).

> **Note**: In Firefox, shared workers cannot be shared between documents loaded in private and non-private windows (bug 1177621).

Spawning a shared worker

Spawning a new worker is pretty much the same as with a dedicated worker, but with a different constructor name (see index.html and index2.html) — each one has to spin up the worker using code like the following:

```
var myWorker = new SharedWorker("worker.js");
```

One big difference is that with a shared worker you have to communicate via a `port` object — an explicit port is opened that the scripts can use to communicate with the worker (this is done implicitly in the case of dedicated workers).

The port connection needs to be started either implicitly by use of the `onmessage` event handler or explicitly with the `start()` method before any messages can be posted. Although the multiply.js and worker.js files in the demo currently call the `start()` method, those calls are not necessary since the `onmessage` event handler is being used. Calling `start()` is only needed if the `message` event is wired up via the `addEventListener()` method.

When using the `start()` method to open the port connection, it needs to be called from both the parent thread and the worker thread if two-way communication is needed.

```
myWorker.port.start();  // called in parent thread
```

```
port.start();  // called in worker thread, assuming the port varia
```

Sending messages to and from a shared worker

Now messages can be sent to the worker as before, but the `postMessage()` method has to be invoked through the port object (again, you'll see similar constructs in both multiply.js and square.js):

```
squareNumber.onchange = function() {
  myWorker.port.postMessage([squareNumber.value,squareNumber
  console.log('Message posted to worker');
}
```

Now, on to the worker. There is a bit more complexity here as well (worker.js):

```
self.addEventListener('connect', function(e) { // addEventLi
  var port = e.ports[0];
  port.onmessage = function(e) {
    var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
    port.postMessage(workerResult);
  }
  port.start();  // not necessary since onmessage event hand
});
```

First, we use an `onconnect` handler to fire code when a connection to the port happens (i.e. when the `onmessage` event handler in the parent thread is setup, or when the `start()` method is explicitly called in the parent thread).

We use the `ports` attribute of this event object to grab the port and store it in a variable.

Next, we add a `message` handler on the port to do the calculation and return the result to the main thread. Setting up this `message` handler in the worker thread also implicitly opens the port connection back to the parent thread, so the call to `port.start()` is not actually needed, as noted above.

Finally, back in the main script, we deal with the message (again, you'll see similar constructs in both multiply.js and square.js):

```
myWorker.port.onmessage = function(e) {
  result2.textContent = e.data[0];
  console.log('Message received from worker');
}
```

When a message comes back through the port from the worker, we check what result type it is, then insert the calculation result inside the appropriate result paragraph.

About thread safety

The `Worker` interface spawns real OS-level threads, and mindful programmers may be concerned that concurrency can cause “interesting” effects in your code if you aren't careful.

However, since web workers have carefully controlled communication points with other threads, it's actually very hard to cause concurrency problems. There's no access to non-threadsafe components or the DOM. And you have to pass specific data in and out of a thread through serialized objects. So you have to work really hard to cause problems in your code.

Transferring data to and from workers: further details

Data passed between the main page and workers is **copied**, not shared. Objects are serialized as they're handed to the worker, and subsequently, de-serialized on the other end. The page and worker **do not share the same instance**, so the end result is that **a duplicate** is created on each end. Most browsers implement this feature as structured cloning.

To illustrate this, let's create for didactical purpose a function named `emulateMessage()`, which will simulate the behavior of a value that is *cloned and not shared* during the passage from a `worker` to the main page or vice versa:

```
function emulateMessage (vVal) {
    return eval("(" + JSON.stringify(vVal) + ")");
}

// Tests

// test #1
var example1 = new Number(3);
console.log(typeof example1); // object
console.log(typeof emulateMessage(example1)); // number

// test #2
var example2 = true;
console.log(typeof example2); // boolean
console.log(typeof emulateMessage(example2)); // boolean

// test #3
var example3 = new String("Hello World");
console.log(typeof example3); // object
console.log(typeof emulateMessage(example3)); // string

```

```
// test #4
var example4 = {
    "name": "John Smith",
    "age": 43
};
console.log(typeof example4); // object
console.log(typeof emulateMessage(example4)); // object

// test #5
function Animal (sType, nAge) {
    this.type = sType;
    this.age = nAge;
}
var example5 = new Animal("Cat", 3);
alert(example5.constructor); // Animal
alert(emulateMessage(example5).constructor); // Object
```

A value that is cloned and not shared is called *message*. As you will probably know by now, *messages* can be sent to and from the main thread by using `postMessage()`, and the `message` event's `data` attribute contains data passed back from the worker.

example.html: (the main page):

```
var myWorker = new Worker("my_task.js");

myWorker.onmessage = function (oEvent) {
  console.log("Worker said : " + oEvent.data);
};

myWorker.postMessage("ali");
```

my_task.js (the worker):

```
postMessage("I\'m working before postMessage(\'ali\').");

onmessage = function (oEvent) {
  postMessage("Hi " + oEvent.data);
};
```

The structured cloning algorithm can accept JSON and a few things that JSON can't — like circular references.

Passing data examples

Example #1: Create a generic "asynchronous `eval()`"

The following example shows how to use a worker in order to **asynchronously** execute any JavaScript code allowed in a worker, through `eval()` within the worker:

```
// Syntax: asyncEval(code[, listener])

var asyncEval = (function () {
  var aListeners = [], oParser = new Worker("data:text/javas

  oParser.onmessage = function (oEvent) {
    if (aListeners[oEvent.data.id]) { aListeners[oEvent.data
    delete aListeners[oEvent.data.id];
  };

  return function (sCode, fListener) {
    aListeners.push(fListener || null);
    oParser.postMessage({
      "id": aListeners.length - 1,
      "code": sCode
    });
  };
})();
```

The data URL is equivalent to a network request, with the following response:

```
onmessage = function (oEvent) {
  postMessage({
    "id": oEvent.data.id,
    "evaluated": eval(oEvent.data.code)
  });
}
```

Sample usage:

```
// asynchronous alert message...
asyncEval("3 + 2", function (sMessage) {
    alert("3 + 2 = " + sMessage);
});

// asynchronous print message...
asyncEval("\"Hello World!!!\"", function (sHTML) {
    document.body.appendChild(document.createTextNode(sHTML)
});

// asynchronous void...
asyncEval("(function () {\n\tvar oReq = new XMLHttpRequest()
```

Example #2: Advanced passing JSON Data and creating a switching system

If you have to pass some complex data and have to call many different functions both on the main page and in the Worker, you can create a system like the following.

example.html (the main page):

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8"  />
<title>MDN Example - Queryable worker</title>
<script type="text/javascript">
  /*
    QueryableWorker instances methods:
     * sendQuery(queryable function name, argument to pass 1
     * postMessage(string or JSON Data): see Worker.prototyp
     * terminate(): terminates the Worker
     * addListener(name, function): adds a listener
     * removeListener(name): removes a listener
    QueryableWorker instances properties:
     * defaultListener: the default listener executed only w
  */
  function QueryableWorker (sURL, fDefListener, fOnError) {
    var oInstance = this, oWorker = new Worker(sURL), oListe
    this.defaultListener = fDefListener || function () {};
    oWorker.onmessage = function (oEvent) {
      if (oEvent.data instanceof Object && oEvent.data.hasOw
        oListeners[oEvent.data.vo42t30].apply(oInstance, oEv
      } else {
        this.defaultListener.call(oInstance, oEvent.data);
      }
    };
    if (fOnError) { oWorker.onerror = fOnError; }
    this.sendQuery = function (/* queryable function name, a
      if (arguments.length < 1) { throw new TypeError("Query
      oWorker.postMessage({ "bk4e1h0": arguments[0], "ktp3fm
    };
    this.postMessage = function (vMsg) {
      oWorker.postMessage(vMsg);
    };
    this.terminate = function () {
      oWorker.terminate();
    };
    this.addListener = function (sName, fListener) {
      oListeners[sName] = fListener;
    };
    this.removeListener = function (sName) {
      delete oListeners[sName];
    };
  };

  // your custom "queryable" worker
  var oMyTask = new QueryableWorker("my_task.js" /* , yourDe

  // your custom "listeners"

  oMyTask.addListener("printSomething", function (nResult) {
    document.getElementById("firstLink").parentNode.appendCh
  });

  oMyTask.addListener("alertSomething", function (nDeltaT, s
    alert("Worker waited for " + nDeltaT + " " + sUnit + " :
  });
</script>
</head>
<body>
  <ul>
    <li><a id="firstLink" href="javascript:oMyTask.sendQuery
    <li><a href="javascript:oMyTask.sendQuery('waitSomething
    <li><a href="javascript:oMyTask.terminate();">terminate(
  </ul>
</body>
</html>
```

my_task.js (the worker):

```
// your custom PRIVATE functions

function myPrivateFunc1 () {
  // do something
}

function myPrivateFunc2 () {
  // do something
}

// etc. etc.

// your custom PUBLIC functions (i.e. queryable from the mai

var queryableFunctions = {
  // example #1: get the difference between two numbers:
  getDifference: function (nMinuend, nSubtrahend) {
      reply("printSomething", nMinuend - nSubtrahend);
  },
  // example #2: wait three seconds
  waitSomething: function () {
      setTimeout(function() { reply("alertSomething", 3, "se
  }
};

// system functions

function defaultQuery (vMsg) {
  // your default PUBLIC function executed only when main pa
  // do something
}

function reply (/* listener name, argument to pass 1, argume
  if (arguments.length < 1) { throw new TypeError("reply - n
  postMessage({ "vo42t30": arguments[0], "rnb93qh": Array.pr
}

onmessage = function (oEvent) {
  if (oEvent.data instanceof Object && oEvent.data.hasOwnPro
    queryableFunctions[oEvent.data.bk4e1h0].apply(self, oEve
  } else {
    defaultQuery(oEvent.data);
  }
};
```

It is possible to switch the content of each mainpage -> worker and worker -> mainpage message.

Passing data by transferring ownership (transferable objects)

Google Chrome 17+ and Firefox 18+ contain an additional way to pass certain types of objects (transferable objects, that is objects implementing the `Transferable` interface) to or from a worker with high performance. Transferable objects are transferred from one context to another with a zero-copy operation, which results in a vast performance improvement when sending large data sets. Think of it as pass-by-reference if you're from the C/C++ world. However, unlike pass-by-reference, the 'version' from the calling context is no longer available once transferred. Its ownership is transferred to the new context. For example, when transferring an `ArrayBuffer` from your main app to a worker script, the original `ArrayBuffer` is cleared and no longer usable. Its content is (quite literally) transferred to the worker context.

```
// Create a 32MB "file" and fill it.
var uInt8Array = new Uint8Array(1024*1024*32); // 32MB
for (var i = 0; i < uInt8Array.length; ++i) {
  uInt8Array[i] = i;
}

worker.postMessage(uInt8Array.buffer, [uInt8Array.buffer]);
```

Note: For more information on transferable objects, performance, and feature-detection for this method, read Transferable Objects: Lightning Fast! on HTML5 Rocks.

Embedded workers

There is not an "official" way to embed the code of a worker within a web page, like `<script>` elements do for normal scripts. But a `<script>` element that does not have a `src` attribute and has a `type` attribute that does not identify an executable mime-type can be considered a data block element that JavaScript could use. "Data blocks" is a more general feature of HTML5 that can carry almost any textual data. So, a worker could be embedded in this way:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>MDN Example - Embedded worker</title>
<script type="text/js-worker">
  // This script WON'T be parsed by JS engines because its m
  var myVar = "Hello World!";
  // Rest of your worker code goes here.
</script>
<script type="text/javascript">
  // This script WILL be parsed by JS engines because its mi
  function pageLog (sMsg) {
    // Use a fragment: browser will only render/reflow once.
    var oFragm = document.createDocumentFragment();
    oFragm.appendChild(document.createTextNode(sMsg));
    oFragm.appendChild(document.createElement("br"));
    document.querySelector("#logDisplay").appendChild(oFragm
  }
</script>
<script type="text/js-worker">
  // This script WON'T be parsed by JS engines because its m
  onmessage = function (oEvent) {
    postMessage(myVar);
  };
  // Rest of your worker code goes here.
</script>
<script type="text/javascript">
  // This script WILL be parsed by JS engines because its mi

  // In the past...:
  // blob builder existed
  // ...but now we use Blob...:
```

```
  var blob = new Blob(Array.prototype.map.call(document.quer

  // Creating a new document.worker property containing all
  document.worker = new Worker(window.URL.createObjectURL(bl

  document.worker.onmessage = function (oEvent) {
    pageLog("Received: " + oEvent.data);
  };

  // Start the worker.
  window.onload = function() { document.worker.postMessage("
</script>
</head>
<body><div id="logDisplay"></div></body>
</html>
```

The embedded worker is now nested into a new custom `document.worker` property.

Further examples

This section provides further examples of how to use web workers.

Performing computations in the background

Workers are mainly useful for allowing your code to perform processor-intensive calculations without blocking the user interface thread. In this example, a worker is used to calculate Fibonacci numbers.

The JavaScript code

The following JavaScript code is stored in the "fibonacci.js" file referenced by the HTML in the next section.

```
var results = [];

function resultReceiver(event) {
  results.push(parseInt(event.data));
  if (results.length == 2) {
    postMessage(results[0] + results[1]);
  }
}
```

```

function errorReceiver(event) {
  throw event.data;
}

onmessage = function(event) {
  var n = parseInt(event.data);

  if (n == 0 || n == 1) {
    postMessage(n);
    return;
  }

  for (var i = 1; i <= 2; i++) {
    var worker = new Worker("fibonacci.js");
    worker.onmessage = resultReceiver;
    worker.onerror = errorReceiver;
    worker.postMessage(n - i);
  }
 };
```

The worker sets the property `onmessage` to a function which will receive messages sent when the worker object's `postMessage()` is called (note that this differs from defining a global *variable* of that name, or defining a *function* with that name. `var onmessage` and `function onmessage` will define global properties with those names, but they will not register the function to receive messages sent by the web page that created the worker). This starts the recursion, spawning new copies of itself to handle each iteration of the calculation.

The HTML code

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"  />
    <title>Test threads fibonacci</title>
  </head>
  <body>

  <div id="result"></div>

  <script language="javascript">

    var worker = new Worker("fibonacci.js");

    worker.onmessage = function(event) {
      document.getElementById("result").textContent = event.
      dump("Got: " + event.data + "\n");
    };

    worker.onerror = function(error) {
      dump("Worker error: " + error.message + "\n");
      throw error;
    };

    worker.postMessage("5");

  </script>
  </body>
</html>
```

The web page creates a `div` element with the ID `result` , which gets used to display the result, then spawns the worker. After spawning the worker, the `onmessage` handler is configured to display the results by setting the contents of the `div` element, and the `onerror` handler is set to dump the error message.

Finally, a message is sent to the worker to start it.

Try this example.

Performing web I/O in the background

You can find an example of this in the article Using workers in extensions .

Dividing tasks among multiple workers

As multi-core computers become increasingly common, it's often useful to divide computationally complex tasks among multiple workers, which may then perform those tasks on multiple-processor cores.

Other types of worker

In addition to dedicated and shared web workers, there are other types of worker available:

- ServiceWorkers essentially act as proxy servers that sit between web applications, and the browser and network (when available). They are intended to (amongst other things) enable the creation of effective offline experiences, intercepting network requests and taking appropriate action based on whether the network is available and updated assets reside on the server. They will also allow access to push notifications and background sync APIs.
- Chrome Workers are a Firefox-only type of worker that you can use if you are developing add-ons and want to use workers in extensions and have access to js-ctypes in your worker. See `ChromeWorker` for more details.
- Audio Workers provide the ability for direct scripted audio processing to be done in a web worker context.

Functions and interfaces available in workers

You can use most standard JavaScript features inside a web worker, including:

- `Navigator`
- `XMLHttpRequest`
- `Array`, `Date`, `Math`, and `String`
- `Window.requestAnimationFrame`, `WindowTimers.setTimeout`, and `WindowTimers.setInterval`

The main thing you *can't* do in a Worker is directly affect the parent page. This includes manipulating the DOM and using that page's objects. You have to do it indirectly, by sending a message back to the main script via `DedicatedWorkerGlobalScope.postMessage`, then actioning the changes from there.

Note: For a complete list of functions available to workers, see Functions and interfaces available to workers.

Specifications

Specification	Status	Comment

WHATWG HTML Living Standard	LS Living Standard	No change from Web Workers.
Web Workers	ED Editor's Draft	Initial definition.

Browser compatibility

Desktop Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	4[1]	3.5 (1.9.1)	10.0	10.6[1]	4[2]
Shared workers	4[1]	29 (29)	Not supported	10.6	5 Not supported 6.1[4]
Passing data using structured cloning	13	8 (8)	10.0	11.5	6
Passing data using transferable objects	17 webkit 21	18 (18)	Not supported	15	6
Global `URL`	10[3] 23	21 (21)	11	15	6[3]

[1] Chrome and Opera give an error "`Uncaught SecurityError: Failed to construct 'Worker': Script at 'file:///Path/to/worker.js' cannot be accessed from origin 'null'.`" when you try to run a worker locally. It needs to be on a proper domain.

[2] As of Safari 7.1.2, you can call `console.log` from inside a worker, but it won't print anything to the console. Older versions of Safari don't allow you to call `console.log` from inside a worker.

[3] This feature is implemented prefixed as `webkitURL`.

[4] Safari removed SharedWorker support.

See also

- `Worker` interface

- `SharedWorker` interface
- Functions available to workers
- Advanced concepts and examples