WEB WORKERS

Multi-threading in JavaScript

Java is a single-threaded language. Your own code shares the same thread as the UI.

This means that heavy computation in your code can effect the speed of the UI, or other parts of your own code as they all compete for the processor.

To make your applications/pages more responsive, or to speed up certain parts of your code you can have them execute in a separate thread.

You can do this with Web Workers.

The code you want to execute in the new thread (the Web Worker) must be placed in a separate **.js** file of its own.

In the following example we will create a text file called **my_worker.js** and put the code for the web worker there.

In your main page (**index.html**) - i.e. in your application JavaScript code (running the the main UI thread) - you create a **Worker** object by passing it the filename of your web worker .js file. This **Worker** object will allow us communicate with the web worker's thread.

```
var ww = new Worker("my_worker.js");
```

Main JavaScript File

index.html

NOTE: In the following slides the key in the corner will indicate whether the code you are seeing is in the main JavaScript file or the Web Worker file.

```
var ww = new Worker("my_worker.js");
```

Main JavaScript File

index.html

In the interests of backwards compatibility & graceful degradation you could check if the browser supports Web Workers before trying to use them.

```
if (window.Worker)
{
  var ww = new Worker("my_worker.js");

 <code using the Web Worker>

}
```

Main JavaScript File

index.html

To have the Web Worker execute code in its thread we can send it a message with the **postMessage()** method.

```
if (window.Worker)
{
  var ww = new Worker("my_worker.js");

  ww.postMessage("some data");
}
```

Main JavaScript File

index.html

Inside the Web Worker file we can set up an event handler (**onmessage**) for incoming messages.

The event handler will be executed whenever the main thread (in **index.html**) calls **postMessage()** on the corresponding **Worker** object.

```
onmessage = function(e){

      <code to execute when a message is received>


}
```

Web Worker File

my_worker.js

The code in the web worker can be executed while the main thread is also running.

There are some limitations to what you can and can't do.

E.g.

> **You cannot access the DOM**
>
> **You don't have access to the window object** (although you do have access to some of its properties).
>
> You **can use** the **XMLHttpRequest** object but **can't use** its **responseXML** property.

window is not the global object in a Web Worker.

Instead you have access to **WorkerGlobalObject** (which shares some of the features of window such as the **console**, **indexedDB**, etc.).

E.g.

Web Worker File

```
    console.log("Testing");
```

Inside the web worker file this accesses the console property of the **WorkerGlobalObject** (and not window). The effect is the same though.

Since you can't access the DOM directly from the Web Worker you can communicate with the main thread which does have access to the DOM.

E.g. the main thread can have a Web Worker perform some intensive calculations, and when finished, the Worker can send the completed result back to the main thread which can display it with the DOM.

If the web worker wants to communicate with the original thread it can also post a message.

```
onmessage = function(e){
        <code to execute when a message is received>
        postMessage("done");
}
```

Web Worker File

my_worker.js

The original thread can also listen for incoming messages.

```
if (window.Worker)
{
    var ww = Worker();

    ww.postMessage("some data");

    ww.onmessage = function(e)
    {
        <code to execute when a message is received>
    }
}
```

Main JavaScript File

index.html

The data sent from the Worker can be found in the **data** property of the **event** object received by the **onmessage** event handler. .

```
if (window.Worker)
{
    var ww = Worker();

    ww.postMessage("some data");

    ww.onmessage = function(e)
    {
        console.log(e.data)
    }
}
```

JavaScript will copy a message event into the handler's parameter

The argument passed to **postMessage()** in the Web Worker will be found in the event's **data** property.

Main JavaScript File

index.html

You can only pass **one parameter** in **postMessage()**.

If you want to pass more than one piece of data you can **pass an array or object**.

Here we pass back the contents of 3 variables in one message by using an array.

```
onmessage = function(e){
    var data1, data2, data3;
    <code to execute when a message is received>
    postMessage([data1, data2, data3]);
}
```

Web Worker File

my_worker.js

And here we use an object.

```
onmessage = function(e){

    var data1, data2, data3;

    <code to execute when a message is received>

    postMessage({name: data1, address: data2, id: data3});
}
```

Web Worker File

my_worker.js

On the receiving side you can access the array/object as you normally would.

Receiving an Array

```
if (window.Worker)
{
    var ww = Worker();

    ww.postMessage("some data");

    ww.onmessage = function(e)
    {
        console.log(e.data[1])
    }
}
```

Main JavaScript File

index.html

On the receiving side you can access the array/object as you normally would.

Receiving an Object

```
if (window.Worker)
{
    var ww = Worker();

    ww.postMessage("some data");

    ww.onmessage = function(e)
    {
        console.log(e.data.name)
    }
}
```

Main JavaScript File

index.html

You can also send data to the Worker in arrays or objects.

Sending an Array

```
if (window.Worker)
{
    var ww = Worker();

    ww.postMessage(["some data", "Some more data"]);

    ww.onmessage = function(e)
    {
        console.log(e.data.name)
    }
}
```

Main JavaScript File

index.html

It is important to note that **objects** passed to and from Web Workers are **cloned**.

I.e. if you are sending an object in a message, it is copied and the Worker uses the copy and not the original.

Therefore, changes made to an object in a Web Worker will have no effect on the version in the main thread (and vice versa).

Example

This simple example does a loop in the Web Worker, sending back the counter variable to the main thread so it can display it on the UI.

(If you tried this entirely in the UI thread, the UI would not be refreshed during the loop as the loop itself monopolises the thread).

```
onmessage = function(e){

   for (var i = 0; i < 20000000; i++) {

      postMessage (i);

   };
}
```

Once this web worker receives a message (any message in this case since we don't check) it starts a loop and sends the loop counter variable back to the main thread.

Web Worker File

```
<script>

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

// Start off the worker code by sending an empty message
worker.postMessage(null);

worker.onmessage = function(e)
{
  document.getElementById("output").innerHTML = e.data;
};

</script>

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

The main JavaScript file displays any message it receives from the Web Worker.

Main JavaScript File

We can improve the performance by sending fewer message between the the main thread and the web worker. We can do this since we couldn't possibly see all the numbers anyway the go by so quickly.

In the following modification of the web worker we only send every 1000th number.

```
onmessage = function(e){


  for (var i = 0; i < 20000000; i++) {

    if (i % 1000)
   {
        postMessage (i);
   }

  };
}
```

We only send the counter variable if it evenly divisible by 1000. I.e. we only send every 1000th number.

Web Worker File

importScripts()

Since you don't have access to the DOM in a web worker and can't use <script> tags, you can include external scripts by using the **importScripts()** function.

You can list as many scripts as you need (or, alternatively, you could use multiple **importScript()** calls)

```
    importScripts("file1.js", "file2.js");
```

They will be executed in the order specified above.

Misc

Note 1: Web Workers can also spawn sub workers of their own.

Note 2: A single web worker can be shared by several other files.

Note 3: For security reasons some browsers (e.g. Chrome) won't allow you to use web workers locally (you will need to server them from a web server)