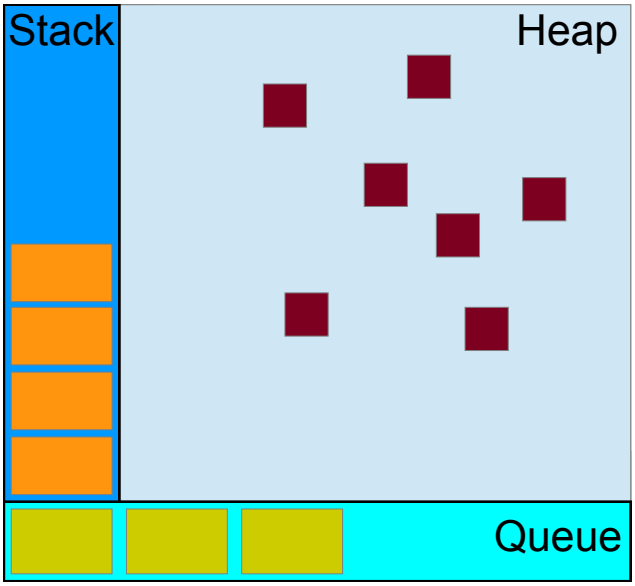
Concurrency model and Event Loop

JavaScript has a concurrency model based on an "event loop". This model is quite different than the model in other languages like C or Java.

Runtime concepts

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

Visual representation



Stack

Function calls form a stack of *frames*.

```
1 function f(b){
```

```
  var a = 12;
  return a+b+35;
}

function g(x){
  var m = 4;
  return f(m*x);
}

g(21);
```

When calling `g`, a first frame is created containing `g` arguments and local variables. When `g` calls `f`, a second frame is created and pushed on top of the first one containing `f` arguments and local variables. When `f` returns, the top frame element is popped out of the stack (leaving only `g` call frame). When `g` returns, the stack is empty.

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 contains a message queue, which is a list of messages to be processed. To each message is associated a function. When the stack is empty, a message is taken out of the queue and processed. The processing consists of calling the associated function (and thus creating an initial stack frame). The message processing ends when the stack becomes empty again.

Event loop

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

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

`queue.waitForMessage` waits synchronously for a message to arrive if there is none currently.

"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 pre-empted 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 can be stopped at any point 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 added any time 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.

Calling `setTimeout` will add a message to the queue after the time passed as second argument. If there is no other message in the queue, the message is processed right away; however, if there are messages, the `setTimeout` message will have to wait for other messages to be processed. For that reason the second argument indicates a minimum time and not a guaranteed time.

Zero delays

Zero delay doesn't actually 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 awaiting 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, but not a guaranteed time.

```
(function () {

  console.log('this is the start');

  setTimeout(function cb() {
    console.log('this is a msg from call back');
  });

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

  setTimeout(function cb1() {
    console.log('this is a msg from call back1');
```

```
  }, 0);

  console.log('this is the  end');

})();
```

Several Runtime 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 an XHR request to return, it can still process other things like user input.

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