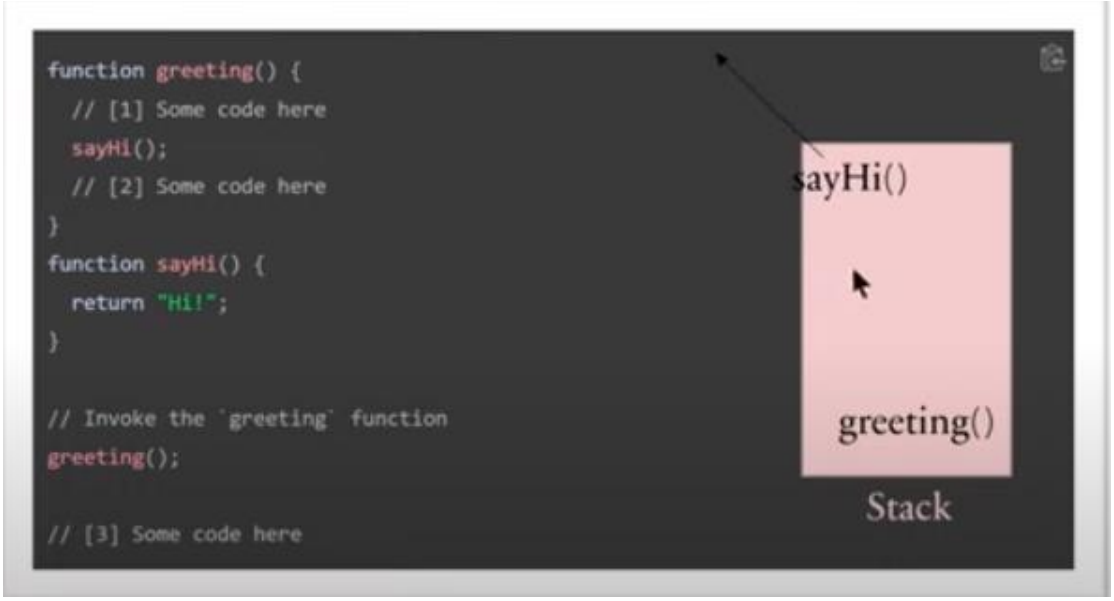
The event loop

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.

- JavaScript is a single threaded, non blocking, asynchronous concurrent language.
- V8 is a javascript runtime which has a callstack and heap.
- It has call stack, an event loop and a callback queue + other APIs
- Heap is used for memory allocation and stack holds execution context.
- DOM, setTimeout, XML HttpRequest don't exist in V8 source code.

CallStack



Asynchronous Callback

- Sometimes the JavaScript code can take a lot of time and this can block the page re render.
- JavaScript has asynchronous callbacks for non blocking behaviour.
- JavaScript runtime can do only one thing at a time.

- Browser gives us other things which work along with the runtime web APIs.
- In node.js these are available as C++ APIs.

Task Queue

- JavaScript can do only one thing at a time.
- The rest are queued to the task queue waiting to be executed.
- When we run setTimeout, webapis will run a timer and push the function provided to setTimeout to the task queue once the timer ends.
- These tasks will be pushed to stack where they can executed.

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.
- The event loop pushes the tasks from task queue to the call stack.
- setTimeout(fun1, 0) can be used to defer a function until all pending tasks (so far) have been executed.
- We can see how these things work in action by visiting.

```
setTimeout(function timer() {
      console.log('You clicked the button!');
}, 2000);


console.log("Hi!");

setTimeout(function timeout() {
    console.log("Click the button!");
}, 5000);

console.log("Welcome to loupe.");
```

Output

```
Hi!
Welcome to loupe.
You clicked the button!
Click the button!
```

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.

```
(() => {
  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");
})();
```

Output

```
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
```