Chapter 49: The Event Loop

Section 49.1: The event loop in a web browser

The vast majority of modern JavaScript environments work according to an *event loop*. This is a common concept in computer programming which essentially means that your program continually waits for new things to happen, and when they do, reacts to them. The *host environment* calls into your program, spawning a "turn" or "tick" or "task" in the event loop, which then *runs to completion*. When that turn has finished, the host environment waits for something else to happen, before all this starts.

A simple example of this is in the browser. Consider the following example:

```
<!DOCTYPE html>
<title>Event loop example</title>

<script>
console.log("this a script entry point");

document.body.onclick = () => {
  console.log("onclick");
};

setTimeout(() => {
  console.log("setTimeout callback log 1");
  console.log("setTimeout callback log 2");
}, 100);
</script>
```

In this example, the host environment is the web browser.

1. The HTML parser will first execute the `<script>`. It will run to completion.
2. The call to `setTimeout` tells the browser that, after 100 milliseconds, it should enqueue a task to perform the given action.
3. In the meantime, the event loop is then responsible for continually checking if there's something else to do: for example, rendering the web page.
4. After 100 milliseconds, if the event loop is not busy for some other reason, it will see the task that `setTimeout` enqueues, and run the function, logging those two statements.
5. At any time, if someone clicks on the body, the browser will post a task to the event loop to run the click handler function. The event loop, as it goes around continually checking what to do, will see this, and run that function.

You can see how in this example there are several different types of entry points into JavaScript code, which the event loop invokes:

- The `<script>` element is invoked immediately
- The `setTimeout` task is posted to the event loop and run once
- The click handler task can be posted many times and run each time

Each turn of the event loop is responsible for many things; only some of them will invoke these JavaScript tasks. For full details, see the HTML specification

One last thing: what do we mean by saying that each event loop task "runs to completion"? We mean that it is not generally possible to interrupt a block of code that is queued to run as a task, and it is never possible to run code interleaved with another block of code. For example, even if you clicked at the perfect time, you could never get the

above code to log `"onclick"` in between the two `setTimeout callback log 1/2"`s. This is due to the way the task-posting works; it is cooperative and queue-based, instead of preemptive.

Section 49.2: Asynchronous operations and the event loop

Many interesting operations in common JavaScript programming environments are asynchronous. For example, in the browser we see things like

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

and in Node.js we see things like

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

How does this fit with the event loop?

How this works is that when these statements execute, they tell the *host environment* (i.e., the browser or Node.js runtime, respectively) to go off and do something, probably in another thread. When the host environment is done doing that thing (respectively, waiting 100 milliseconds or reading the file `file.txt`) it will post a task to the event loop, saying "call the callback I was given earlier with these arguments".

The event loop is then busy doing its thing: rendering the webpage, listening for user input, and continually looking for posted tasks. When it sees these posted tasks to call the callbacks, it will call back into JavaScript. That's how you get asynchronous behavior!