Asynchronous Programming in Javascript

CSCI 5828: Foundations of Software Engineering
Lectures 29 — 12/08/2015

Credit Where Credit Is Due

- Some of the material in this lecture is drawn from or inspired by
  - You Don't Know Javascript: Async & Performance
    - by Kyle Simpson; Published by O'Reilly
    - Note: The entire "You Don't Know Javascript" series is fantastic!
  - Async Javascript: Build More Responsive Apps with Less Code
    - by Trevor Burnham; Published by Pragmatic Programmers
- Both are excellent books that cover doing things asynchronously in Javascript

Goals

- Discuss the notion of asynchronous programming in Javascript
  - The gap between “now” and “later”
  - The event loop
- Traditional Approach
  - Callbacks
  - “Callback Hell”
- New Approach
  - Promises

Javascript

- Javascript has been the “language of the web” for 20 years
  - past history of incompatible implementations across browsers,
  - recently, however, thanks to ECMA standardization and libraries like JQuery and Underscore, it is much easier to get consistent behavior for web apps across browsers
- On its way to becoming a “full stack” programming language thanks to several open source implementations (most notably Google Chrome)
  - Full stack means that Javascript can be used to develop the entire stack of technologies used for a software system
    - Javascript in user interface (e.g. Angular web application), Javascript in middleware (Node.js), Javascript in persistence layer (MongoDB)

The Challenges of Javascript

- Developers often have trouble with Javascript because of a mismatch between the mental model they impose on the language and its actual model
  - It's easy to think of it as only an imperative, object-oriented language
    - You might expect to be able to do things like traditional subclassing and perhaps multi-threaded programming
    - It is actually a functional/imperative hybrid AND an event-based language that provides object-oriented features via "prototypical inheritance"
    - It is also a "hosted" language in that Javascript never runs in standalone mode; there is always a "container" that Javascript runs "in"
      - The browser (Chrome), a web server (Node.js), a database (Mongo)
- Get any of this wrong in your thinking and you're in for a world of pain

Event Based == Potential for Asynchrony

- The fact that Javascript is event-based means that there is the potential for asynchronous programming
  - Some code is executed **now** and some code is executed *later*
- Consider this complete Javascript program (to be executed by Node.js)
  - `console.log("I like Javascript!");`
- This looks like an imperative program and it acts like one
  - Run it and it will print out "I like Javascript" as expected
- But…

`[imperative.js]`

Under the covers…

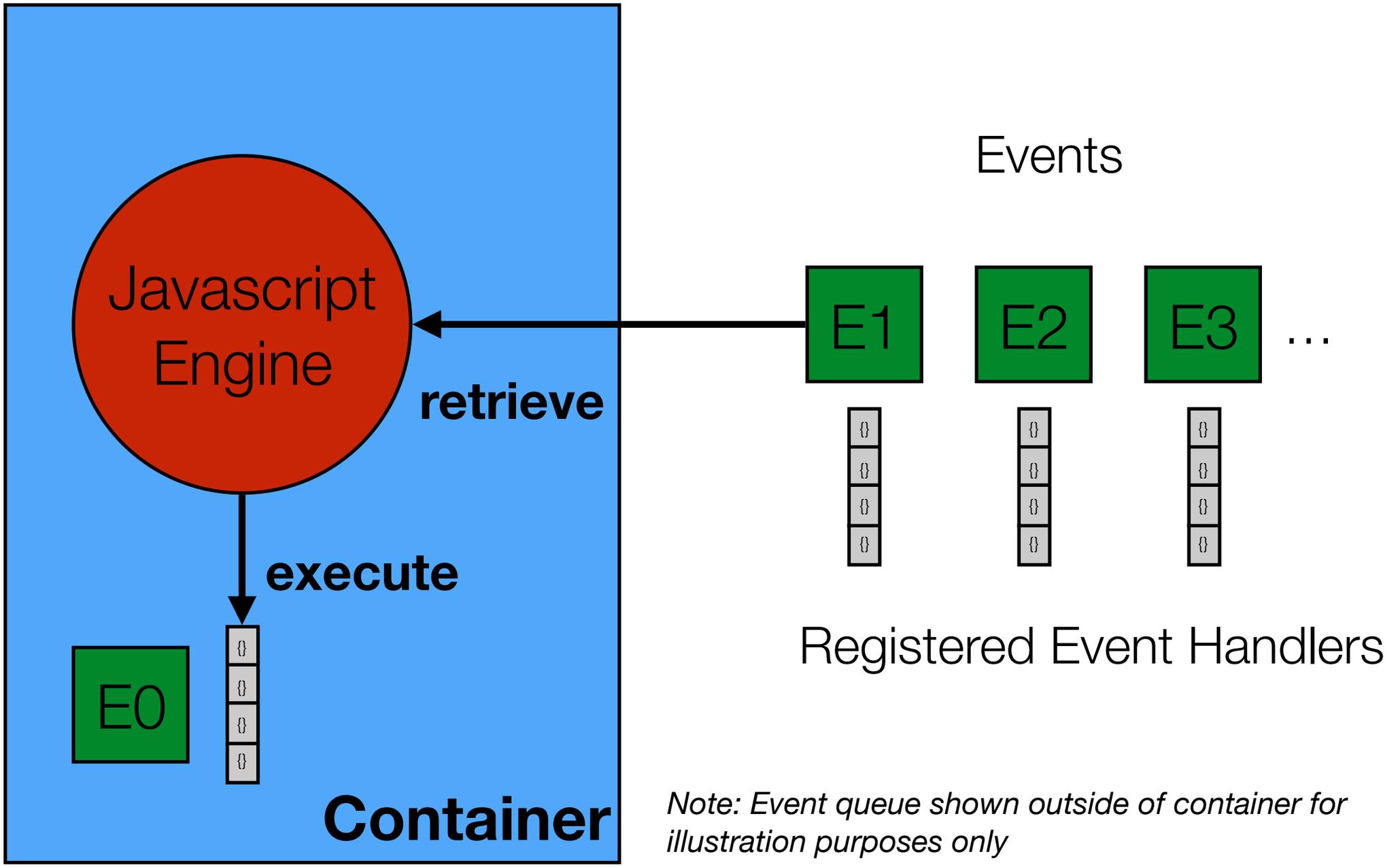
- Our simple program is actually a bit more complex; more like this:
  - `var f = function() {`
    - `console.log("I like Javascript!");`
  - `}`
  - `global.on("program start", f);`
  - «sometime later»
    - `global.emit("program start")`
- That is Node.js
  - 1) takes the contents of the input file, 2) wraps it in a function, 3) associates that function as an **event handler** that is associated with the "start" or "launch" event of the program; 4) performs other initialization, 5) emits the program start event; 6) the event gets added to the event queue; 7) the Javascript engine pulls that event off the queue and executes the registered handler, and then (finally) 8) our program runs!

`[imperative2.js]`

The Implication

- No code runs in Javascript unless
  - an event occurred
  - that event has at least one registered event handler
  - that event has reached the front of the event queue
- If all of that is true, then the Javascript engine will
  - pull the event off the queue
  - retrieve the event handlers
  - invoke them in order (which may add more events to the queue)
  - loop back and process the next event
- Events can also be added to the queue by Javascript's container
  - mouse move events, mouse clicks, network IO, operating system interrupts, etc.

The Event Queue



Note: Event queue shown outside of container for illustration purposes only

The other implication

- Javascript is single-threaded
  - It can only do one thing at a time!
    - Namely: execute the current event handler
- Boon with respect to “concurrency”
  - If you have two event handlers that share a mutable data structure, you don’t have to worry about race conditions
    - only **one** event handler runs at a time
  - It’s not all roses, however; the two event handlers can arbitrarily interleave and if you care about ordering with respect to updating that shared data structure then you still have work to do
    - but, out of the box, you do not have to worry about interleaving reads/writes to the same data structure

Adding Events to the Queue

- One way for Javascript programs to add events to the event queue is via a call to `setTimeout()`
  - `setTimeout()` is a function that accepts two parameters
    - an event handler (i.e. a function), and
      - a delay: the delay before the handler gets added to the event queue
- This causes a timer to be created
  - it executes immediately and waits for the specified delay
  - it then adds a timeout event to the queue with the specified event handler
- The Javascript engine will then process that event as normal

What's the Output?

- Back to the potential for asynchrony
  - In Javascript, if the code executing **now**, adds an event to the event queue (with an associated handler), or registers an event handler for an event that could happen, then some code is going to execute *later*
- What's the output of this program?

```
for (var i = 1; i <= 3; i++) {
 setTimeout(
  function() {
   console.log(i);
  }, 0);
};
```

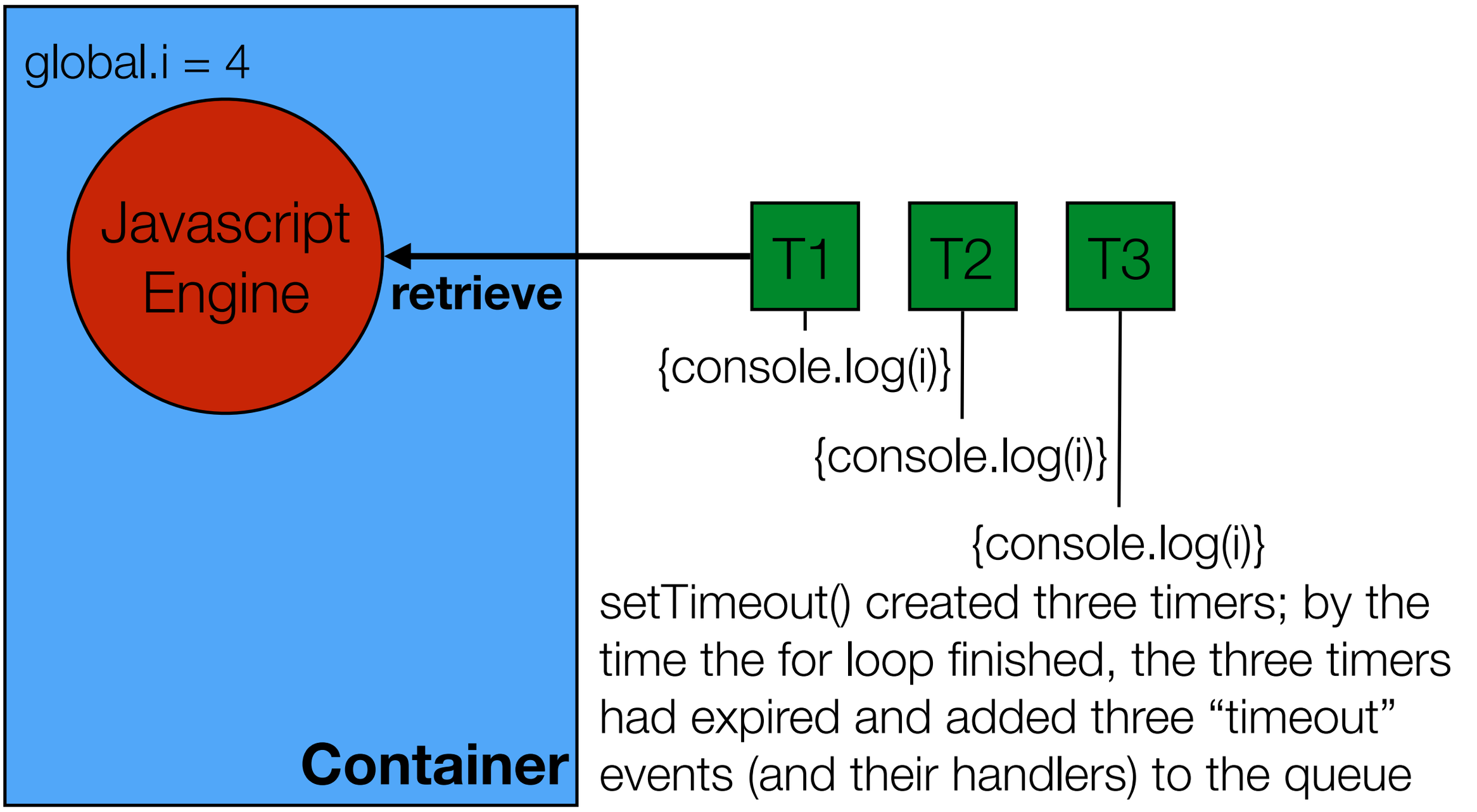
- What part of this program executes now? later?

[async_code.js]

asynchronous code runs “later”

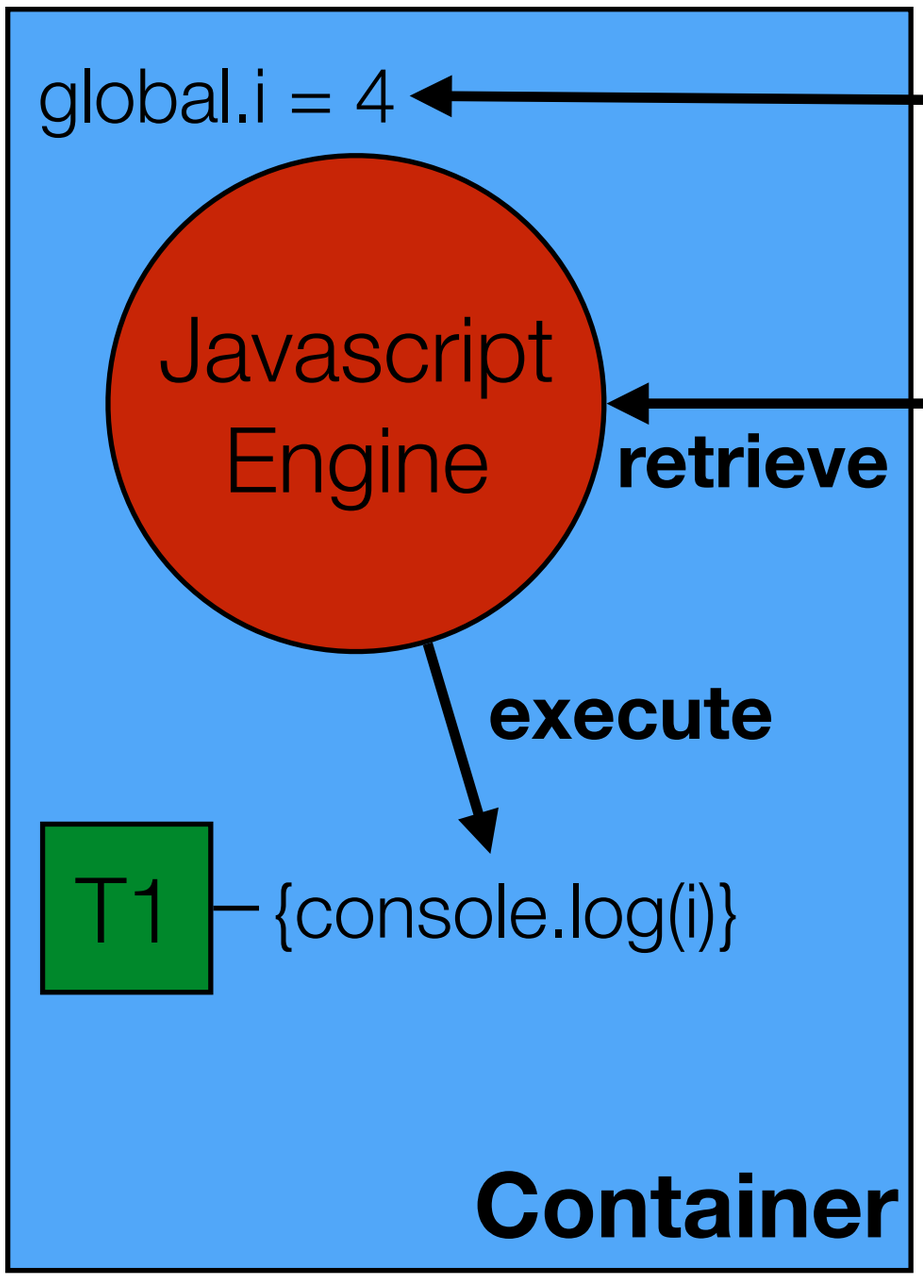
- The output is
  - 4
  - 4
  - 4
- while you might think the code is equivalent to
  - setTimeout(function(){ console.log(1); }, 0);
  - setTimeout(function(){ console.log(2); }, 0);
  - setTimeout(function(){ console.log(3); }, 0);
- It instead is capturing the reference to the variable i, which is then used when the callback runs which is **AFTER** the three calls to setTimeout.

setTimeout() creates code that will run later

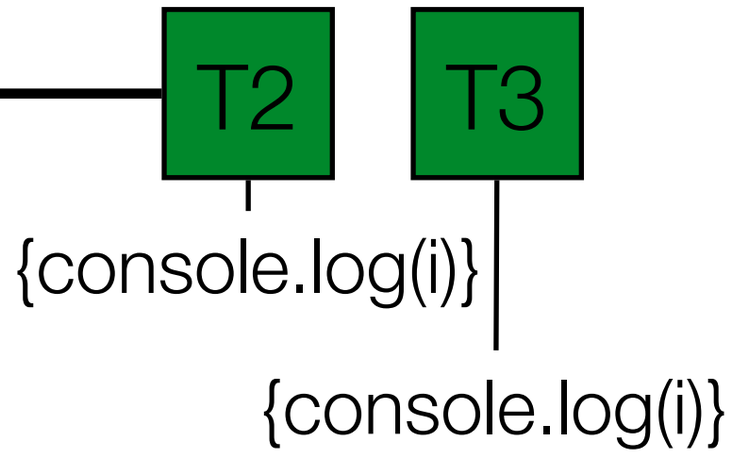


setTimeout() created three timers; by the time the for loop finished, the three timers had expired and added three “timeout” events (and their handlers) to the queue

setTimeout() creates code that will run later



The global.i value is set to 4 by the for loop's use of the ++ operator



When each event handler is executed, it uses its reference to i to print out the current value (4)

One more example

```
var start = new Date;
setTimeout(
   function() {
      var end = new Date;
      console.log('Time elapsed:', end - start, ‘ms');
   }, 500
  );
  while (new Date - start < 1000) {};
```

- What’s the output?

[two_delays.js]

The answer

- You will typically see something like
  - `Time elapsed: 1000 ms`
- This should make sense given what we covered above
- If you expected to see:
  - `Time elapsed: 500 ms`
- then you are most likely applying a “multi-threaded” view to Javascript
  - in other words, you might think that the timer starts counting down and then invokes the “time out” handler concurrently or in parallel; it doesn’t.
  - The timer does indeed run in parallel (it’s being handled by the container, not the Javascript engine) but it’s action is to ADD a “timeout” event to the event queue; that event cannot be processed until the current event is done; in this example *the current event* is **the entire main program**!

How fast is the event loop? (I)

- Not very fast; consider this code

```
var fireCount = 0;
var start = new Date;
var timer = setInterval(function() {
  if (new Date - start > 1000) {
    clearInterval(timer);
    console.log(fireCount);
    return;
  }
  fireCount++;
}, 0);
```

- This code uses setInterval() to create a recurring event
  - however, it specifies 0 as the delay between events
    - so theoretically it should “flood” the event queue for one second
      - it then prints out how many events it created

How fast is the event loop? (II)

- The result? On my machine: 718 events
- Why so slow?
  - It creates a timer
  - loop
    - That timer expires
    - When it does, it adds the “done” event to the queue and the callback
    - The callback gets executed; while it does so, it resets the timer
- That’s a lot of overhead and that overhead slows things down
  - Can it process events more quickly? Yes, but…

How fast is the event loop? (III)

- Many environments provide a mechanism for specifying callbacks that should execute before any I/O is allowed (i.e.
  - This mechanism essentially cuts in line in front of the event queue
    - and your application can become non-responsive if it does this a lot because the event queue is blocked
- BUT, it is much more efficient when speed is of the essence
  - how much more?
- Let's see; for Node.js, this mechanism is called `process.nextTick()`
  - Let's rewrite the example on the previous slide and use `nextTick()` rather than `setInterval()`
    - Note: `setInterval()` is standard across all JavaScript environments; `process.nextTick()` is Node.js specific

How fast is the event loop? (IV)

```
var fireCount = 0;
var start = new Date;
var tick = function() {
  if (new Date - start > 1000) {
    console.log(fireCount);
    return;
  }
  fireCount++;
  process.nextTick(tick)
}
process.nextTick(tick)
```

- The result? On my machine: 3576104 (!)
  - 3.57M handlers executed in one second vs 718!

Callbacks

- The traditional way of handling asynchronous events in Javascript is by using a callback
  - A callback is simply a function that is registered as the event handler for some event we know will happen in the future
- For instance, Node.js provides an API to the local file system
  - it has synchronous versions of these API methods, but these are frowned upon because
    - I/O can take a lot of time and if that time *happens in an event handler* the **whole event queue is blocked**!
- Instead, it is recommended that you use the asynchronous versions of these file system methods
  - When you ask one of these methods to do something, you register a callback to allow it to notify you when it is done

Example (I)

```
var fs = require("fs");
fs.readdir('.', function(err, filenames) {
   if (err) throw err;
   console.log("Number of Directory Entries:", filenames.length)
   filenames.forEach(function (name) {
      fs.stat(name, function(err, stats) {
         if (err) throw err;
         var result = stats.isFile() ? "file" : "directory";
         console.log(name, "is a", result);
      });
   });
});
```

- Here we use two separate file system routines: `readdir` and `stat`
  - It is important to learn how to read this code in terms of "now" and "later"

[file_info.js]

Example (II)

```
var fs = require("fs");
fs.readdir('.', …)
```

- In the first "event handler" the code above is executed to completion
  - This is what happens "now". All the rest happens "later"
- The whole main program is revealed to be a call to `fs.readdir()`!
  - :-)
- The second argument to `readdir()` is a function that acts as the callback
  - `readdir()` will go and read the file system; when it is done, it will add the "done" event to the event queue and set the provided callback to be that event's event handler
    - when that event is the first in the queue, the Javascript engine will execute it

Example (III)

```
    function(err, filenames) {
1 =>   if (err) throw err;
2 =>   console.log("Number of Directory Entries:", filenames.length)
3 =>   filenames.forEach(function (name) {
4 =>     fs.stat(name, …);
       });
    });
```

- Here is the callback for `readdir()`; it has 4 lines that execute to completion
  - with the fourth line executed once per element in the current directory in the order provided by the `filenames` array (lexicographic)
  - Note: for Node.js, callbacks take an error object as their first parameter and then a results object as their second parameter
    - thus `({object}, null)` indicates error while `(null, {object})` indicates success
    - This is just a convention and can be different across APIs and systems

Example (IV)

```
function(err, stats) {
  if (err) throw err;
  var result = stats.isFile() ? "file" : "directory";
  console.log(name, "is a", result);
}
```

- Here is the callback for `stat()`
  - It is invoked once per element in the dir with info (stats) about each one
- The order in which each callback executes **cannot be predicted**;
  - it all depends on how long the call to "stat" takes;
    - once that call completes, it adds the "done" event to the queue
  - so the files in our output **do not** appear in lexicographic order;
    - they appear in the order that stat completed

Important Point

- There is a famous quote about Node.js
  - "Everything runs in parallel except your code"
- We just saw an example of that with the previous code
  - When we looped over every element in filenames (sequentially), we called fs.stat() for each file/dir name
    - All of those stat commands execute ***in parallel***
      - As they finish, they ADD events to the queue with your callback attached
        - Each of those callbacks run sequentially
      - Since the I/O commands run in parallel, some finish before others and this leads to the file's reporting "out of order"

Complexity (I)

- The previous example was relatively simple
  - all we did was print information out to standard out
- Imagine if we wanted to collect all of the file sizes of each file and then report on the total size
  - We would need a way to figure out when we were done and then sum up the values and report the final result

Complexity (II)

- If you've never done this before, you might try:

```
var fs = require("fs");
var total = 0
fs.readdir('.', function(err, filenames) {
   if (err) throw err;
   console.log("Number of Directory Entries:", filenames.length)
   filenames.forEach(function (name) {
      fs.stat(name, function(err, stats) {
         if (err) throw err;
         total += stats.size;
      });
   });
});
console.log("Total size:", total);
```

- What's the problem with this approach?

[file_total.js]

Complexity (III)

- The output for one of my directories is:
  - `Total size: 0`
  - `Number of Directory Entries: 20`
- The code to print the total is executed before `readdir()` and `stat()` have had a chance to do their job.
  - We need to wait until they are done before we can print out the result
    - unfortunately, we don't know **WHEN** they will be done
- So, we need to develop an approach that keeps track of our global state asynchronously and then print out the total after all files have been processed
  - There's only one problem: complexity! *(Hello, complexity, my old friend.)*
  - For example, here's one approach we can try… (next slide)

```
var fs = require('fs');

var all_done = function(size) {
  console.log("Total size:", size);
}

var handleFile = function(stats, i, filenames, total) {
  if (i === filenames.length - 1) {
    all_done(total + stats.size);
  } else {
    processFile(i+1, filenames, total+stats.size);
  }
}

var handleDir = function(i, filenames, total) {
  if (i === filenames.length - 1) {
    all_done(total);
  } else {
    processFile(i+1, filenames, total);
  }
}

var processFile = function(i, filenames, total) {
  var name = filenames[i];
  fs.stat(name, function(err, stats) {
    if (err) throw err;
    if (stats.isFile()) {
      handleFile(stats, i, filenames, total);
    } else {
      handleDir(i, filenames, total);
    }
  });
}

fs.readdir('.', function(err, filenames) {
  if (err) throw err;
  console.log("Number of Directory Entries:", filenames.length)
  processFile(0, filenames, 0);
});

```

Complexity (IV)

Holy Wow!

Complexity indeed!

[file_total_fixed1.js]

```
var fs = require('fs');

var all_done = function(size) {
  console.log("Total size:", size);
}

var handleFile = function(stats, i, filenames, total) {
  if (i === filenames.length - 1) {
    all_done(total + stats.size);
  } else {
    processFile(i+1, filenames, total+stats.size);
  }
}

var handleDir = function(i, filenames, total) {
  if (i === filenames.length - 1) {
    all_done(total);
  } else {
    processFile(i+1, filenames, total);
  }
}

var processFile = function(i, filenames, total) {
  var name = filenames[i];
  fs.stat(name, function(err, stats) {
    if (err) throw err;
    if (stats.isFile()) {
      handleFile(stats, i, filenames, total);
    } else {
      handleDir(i, filenames, total);
    }
  });
}

fs.readdir('.', function(err, filenames) {
  if (err) throw err;
  console.log("Number of Directory Entries:", filenames.length)
  processFile(0, filenames, 0);
});

```

Complexity (V)

- The approach: generate one callback per file in the array NOT ALL AT ONCE (which is what we did before) but one after the other
- Each callback generates the next callback
- End when all files have been processed.

It works, but…

- So, this approach works
  - that is, it calculates the correct answer and it's asynchronous
    - for instance, in a web application, we could kick off this code on a directory and be processing other events (clicks, network I/O, animation, web requests, etc.) while it finishes
      - our application stays responsive even as it processes lots of events
- but it is hugely complex
  - especially, if you compare this to synchronous code that does the same thing (see `file_total_sync.js` for an example)
- The complexity of callbacks has a general purpose name: callback hell
  - Let's dig deeper into what this means

The Pyramid of Doom

- One aspect of callback hell comes from the way callback code can look

```
• fs.readdir(…, function(err, result) {
   • fs.stat(…, function(err, result) {
      • fs.readFile(…, function(err, result) {
         • …
      • });
   • });
• });
```

- Each new callback causes another level of indentation
    - And this is typically the “success path”
        - It gets more complicated if you want to handle errors!
- As the You Don’t Know Javascript book says: *callback hell is more than this!*

Thinking in Terms of Now and Later (I)

- In this code snippet, assume that any function that takes a function treats that function like a callback: it adds it to the event queue to execute later
- In what order do the following methods execute?

```
doA(function() {
  doB();
  doC(function() {
    doD();
  });
  doE();
});
doF();
```

Thinking in Terms of Now and Later (II)

```
doA(function() {
  doB();
  doC(function() {
    doD();
  });
  doE();
});
doF();
```

- The answer?
  - `doA(); doF(); doB(); doC(); doE(); doD();`
- The calls to A and F happen first; then the callback to A gets executed, calling B, C, and E. Finally, the callback to C executes and D gets called

Thinking in Terms of Now and Later (III)

- This inability to easily determine the order of execution is another aspect of callback hell
  - You have to always think clearly about what executes now and what causes events to get added to the queue to execute later
- More insidious, sometimes the SAME function will execute a callback either synchronously or asynchronously depending on the context! For example:

```
function foo(callback) {
 if (globalVar > 50) {
  process.nextTick(callback); // asynchronous
 } else {
  callback(); // synchronous
 }
}
```

In both cases, your callback is called; but if your callback calls `foo()` *then you're in trouble, especially in the synchronous case!*

Hardcoded Workflows

- We alluded to this problem on the Pyramid of Doom slide
  - What happens if something goes wrong in your workflow?

```
doA(function() {
  doB();
  doC(function() {
    doD();
  });
  doE();
});
doF();
```

  - Consider what would happen if doB() failed? Who gets notified? doA() is long gone. Should we try again? Who tries again? Should we stop the whole program? What if we should doC() but only conditionally? What happens if doE() fails in this case? Should we perform doC() again?

What a mess!

Issues of Trust

- The final aspect of callback hell concerns trust
  - I'm calling foo() and passing it a callback
- In that case, I'm trusting that foo() will eventually call my callback
  - If foo() is my own code, then I might have high confidence that this will happen; but what if foo() was written by a third party?
- All sorts of problems might occur
  - The callback is called too early; perhaps I'm still setting things up when the callback arrives
  - The callback is called too late; e.g., a stock price warning message
  - The callback is never called; my code is stuck waiting and waiting
  - An error occurred in foo() but the exception is not propagated
  - foo() calls the callback with the wrong parameters, etc.

How do we address callback hell?

- With a higher-level abstraction called ***promises***
  - And, I *promise* to tell you about promises in the next lecture
    - :-)