Overview

- The JavaScript **event loop** and **run-to-completion** semantics.
- Callbacks for event handlers.
- Pyramid of doom.
- Taming asynchronous code: promises.
- Taming asynchronous code: `async`, `await`.

The Need for Concurrency

- Modern CPUs have clocks in the low GHz. That means individual CPU operations occupy under 1 nanosecond.
- Typically, I/O may take in the order of milliseconds which is around a million times slower than CPU operations.
- Highly inefficient to have CPU wait for an I/O operation to complete.
- Need to concurrently do other stuff while waiting for I/O to complete.

Approaches to Concurrency

Two commonly used approaches to concurrency:

Synchronous Blocking Model: When a program attempts to do I/O, the program blocks until the I/O is completed. To allow concurrency, the operating system will schedule some other activity while waiting for the I/O. The unit of scheduling is usually a **process** or **thread**; leads to the process/thread model used by many current OS's.

Asynchronous Event Model: When a program attempts to do I/O, it merely starts the I/O after registering a handler to handle the I/O completion event. The program continues running while the I/O is happening concurrently. The completion of the I/O results in a event which results in the registered handler being called.

JavaScript uses the asynchronous event model.

JavaScript Event Loop

The top-level JavaScript runtime consists of an event loop which pulls extant events off an event queue and calls their registered handlers:

```
while (eventQueue.notEmpty()) {
  const event = eventQueue.remove();
  const handler = event.handler();
  handler.call(); //pass suitable arguments
}
//terminate program
```

- The `hander.call()` **runs to completion**.
- Code does not need to deal with an event handler being interrupted.
- Code still needs to deal with the fact that the order of running of event handlers is not defined.

Run to Completion Consequences

In run-to-completion.js:

```
#!/usr/bin/env nodejs

//BAD CODE!!
function sleep(seconds) {
  const stop = Date.now() + seconds*1000;
  while (Date.now() < stop) {
    //busy waiting: yuck!
  }
}

setTimeout(() => console.log('timeout'),
           1000 /*delay in milliseconds*/);

sleep(5);
console.log('sleep done');
```

Run to Completion Log

```
12:02:17|master/code $ ./run-to-completion.js
sleep done
timeout
12:02:23|master/code $
```

Because of run-to-completion semantics, it will **always** be the case that the *sleep done* message will be output before the *timeout* message.

Why This Concurrency Model

- JavaScript was designed as a language which should be easy for inexperienced programmers to use for scripting dynamic behavior in browsers.
- Browser reacts to user actions by generating events like key-press, mouse-click, etc.
- Browser programmer needs to provide optional event handlers for these events in order to implement browser dynamic behavior.
- Since every event handler runs to completion, programmer can simply concentrate on code for that event, ignoring other events (at least for independent events).
- No need for the programmer to understand complex process / threading models.

Playing with Asynchronous Functions

Use later to run function asynchronously after a random delay:

```
const MAX_TIMEOUT = 3;

function later(fn, ...args) {
  const timeout =
    Math.floor(Math.random()*(MAX_TIMEOUT + 1));
  setTimeout(fn, timeout*1000, ...args);
}
```

```
> .load later.js
undefined
> > later(() => console.log('done'))
undefined
> done //note prompt output before 'done'
```

Using Return Value of Asynchronous Function

```
> function f() {
..... later(() => { console.log('f run'); return 42; });
..... }
undefined
> f()
undefined
> f run

>
```

How do I get hold of the 42 return value.

Return Value of Asynchronous Function: Another Attempt

```
> let ret = -1
undefined
> function f() {
... later(() => { console.log('f run'); ret = 42; });
... }
undefined
> f(); console.log(`ret = ${ret}`)
ret = -1
undefined
> f run

> ret
42
>
```

Passing a Handler for Return Value

```
> function f(succFn) {
... later(() => { console.log('f run'); succFn(42); });
... }
undefined
> f((ret) => console.log(`ret = ${ret}`))
undefined
> f run
ret = 42

>
```

Passing Return Value to Another Asynchronous Function

```
>function g(v, fn) {
..... later(() => { console.log(`g(${v})`); fn(2*v); });
..... }
undefined
> f((v) => g(v, (x) => console.log(`g() value = ${x}`)))
undefined
> f run
g(42)
g() value = 84

>
```

Getting out-of-hand!

Aside: Callbacks and Continuations

- Pass every function a continuation which is a function `cc(v)` of one argument.
- When the function would normally return a value `v`, it should simply call `cc(v)`.
- In *continuation-passing style* a function never returns.
- Programming a chain of event handlers is similar to programming with continuations.

Continuation-Passing Factorial

```
//cc is current continuation.
//cc(n): continue as per function cc() with return value n
//Note no return.  //indicates traditional factorial
function contFact(n, cc) { //function fact(n) {
  if (n <= 1) {
    cc(1);                    //return 1;
  }
  else {
    contFact(n - 1,         //return n*fact(n - 1);
             (f1) => cc(n*f1));
  }
}

contFact(5, (f) => console.log(f));
```

Errors

```
//Normal exception catching
> try {
    throw 'throwing';
  } catch (ex) {
    console.log(`caught ${ex}`);
  }
caught throwing
undefined
```

Errors Continued

```
//Exception in Async not caught
> try {
... f(() => {
..... throw { msg: 'thrown' };
..... }) } catch (ex) {
... console.log(`caught ${ex}`);
... }
undefined
> f run
Thrown: [object Object]
>
```

Problems with Callbacks

- A top-level exception handler does not work for asynchronous callbacks since the handler runs before the callback. Hence exceptions occurring within the callback are not caught by the top-level exception handler.
- If a asynchronous function result needs to be further processed by another asynchronous function, then we need to have nested callbacks.
- A chain of callbacks leads to the **pyramid of doom** because of nesting of callbacks.

Promises

- A Promise is an object representing the eventual completion or failure of an asynchronous operation.
- When a function which requires an asynchronous callback as an argument is called, it returns immediately with an object called a *pending* **Promise**. Subsequently, the callbacks can be added to the promise. The callbacks will be called after the promise has been *settled*.

```
let promise = some_call_which_returns_promise(...);
promise.
  then(callback1).
  then(callback2).
  ...
  catch(errorCallback);
```

Promise Advantages

- Promises can be chained; this avoids the *pyramid of doom*.
- Callbacks are never called before completion of current run of js **event loop**.
- Callbacks added using `then` even after completion of the asynchronous operation will still be called.
- `then()` can be called multiple times to add multiple callbacks (called in order of insertion).
- Allows catching errors much more easily using `catch()`; similar to exception handling.
- `then()` can even be chained after a catch.

Promise API

```
new Promise(
  /* executor */
  function(resolve, reject) { ... }
);
```

- Creates a promise.
- `resolve` and `reject` are single argument functions.
- Executor function executed immediately. Usually will start some kind of asynchronous operation which may return some result.
  1. If the async operation succeeds with some result `value`, then the executor function should call `resolve(value)`.
  2. If the async operation fails with some error `err`, then the executor function should call `reject(err)`.

Promise States

Pending	The underlying operation is not yet complete.
Fulfilled	The underlying operation completed successfully.
Rejected	The underlying operation failed.
Settled	The operation completed, the promise is either *fulfilled* or *rejected*.

A promise is settled only once. The state of the promise will not change once it is settled.

Getting Promise Settlement: `then()`

```
somePromise.then(value, err)
```

- Arguments are one argument functions called when `somePromise` is settled; specifically `value` / `err` are called with fulfillment / rejection value depending on settlement.
- Usually `then()` is called with only the `value` argument, with rejection of `somePromise` handled using a `catch()`.
- `then()` itself returns a promise; this allows chaining `then`'s.
  - If the function passed to `then()` returns a value, then the return'd promise fulfills with that value.
  - If the function passed to `then()` throws an error, then the return'd promise rejects with that error.
  - If the function passed to `then()` returns a promise, then the return'd promise has the same settlement as it.

Getting Promise Rejection: `then()`

```
somePromise.catch(err)
```

- `err` is a one argument functions called with the rejection value of promise `somePromise`.
- `catch()` itself returns a promise; this allows continued promise chaining. Return value is similar to that of `then()`.

Playing with Promises

```
> function p(...args) { console.log(...args); }
> p(1, 2)
1 2
> pr = new Promise((resolve, reject) => resolve(22))
Promise {  22, ... }
> pr.then((v) => p(v))
Promise { <pending>, ... }
> 22
//Promise is settled only once
> pr = new Promise((succ) => { succ(42); succ(22); })
Promise { 42, ... }
> pr.then((v) => p(v))
Promise { <pending>, ... }
> 42
```

Playing with Promises: Chaining then()'s

```
> function f(a, b) { p(a); return a * b; }
undefined
> pr = new Promise((resolve) => resolve(22))
Promise { 22, ... }
> pr.then((val) => f(val, 2)).
... then((val) => f(val, 3)).
... then((val) => p(val))
> Promise { <pending>, ... }
> 22
44
132

>
```

Creating Settled Promises

`Promise.resolve(value)` Returns a promise which is already fulfilled with `value`.

`Promise.reject(err)` Returns a promise which is already rejected with `err`.

Playing with Promises: Asynchronous Functions

```
> function f(a, b, ret) {
... p(`${new Date().toTimeString()}: ${a}`);
... setTimeout(() => ret(a*b), 2000);
... }
undefined
> pr = Promise.resolve(22)
> pr.
... then((v) => new Promise((succ) => f(v, 2, succ))).
... then((v) => new Promise((succ) => f(v, 3, succ))).
... then((v) =>
        p(`${new Date().toTimeString()}: ${v}`))
> 10:54:50 GMT-0500 (EST): 22
10:54:52 GMT-0500 (EST): 44
10:54:54 GMT-0500 (EST): 132

>
```

Playing with Promises: Errors

```
> function t() { return new Date().toTimeString(); }
> pr1 = Promise.reject(new Error(t()))
Promise { <rejected>  Error: 11:12:04 ... }
> (node:24159) UnhandledPromiseRejectionWarning: ...
> p(t()); pr1 =
... Promise.reject(new Error(t())); pr1.catch(()=>{})
11:15:36 GMT-0500 (EST)
...
> p(t()); pr1.
... then((v) => p(v)).
... then((v) => p(v)).catch((err)=>p(err))
11:16:10 GMT-0500 (EST)
...
> Error: 11:15:36 GMT-0500 (EST)
...
```

Playing with Promises: Errors Continued

```
> pr1.
... then((v) => p(`got value ${v}`)).
... then((v) => p(`got value ${v}`)).
... catch((e) => { p(`caught ${e}`); return 42; }).
... then((v) => p(`got value ${v}`))
Promise { <pending>, ... }
> caught Error: 11:15:36 GMT-0500 (EST)
got value 42

>
```

Playing with Promises: Errors Continued

then()-chain continues past catch():

```
> Promise.resolve(1).
... then((v) => { p(`then1: ${v}`); return v*2; }).
... then((v) => { p(`then2: ${v}`); return v*2; }).
... catch((e) => p(`caught ${e}`)).
... then((v) => { p(`then3: ${v}`); return v*2; })
Promise { <pending>, ... }
> then1: 1
then2: 2
then3: 4

>
```

A Curried Promise Creator

In curried-promise.js:

```
//Returns a curried function f, such that calling f(a)(b)
//will result in a promise encapsulating the result of an
//async call to fn(a, b) after DELAY_MILLIS.
function curriedPromise(fn) {
  return (a) => (b) =>
    new Promise(function(resolve, reject) {
      setTimeout(function() {
        let value, err;
        try {
          value = fn(a, b);
        }
        catch (e) {
          err = e;
        }
        if (err) reject(err); else resolve(value);
      }, DELAY_MILLIS);
```

Using Curried Promise

```
> .load curried-promise.js
> mul = curriedPromise((a, b) => a*b)
> [mul2, mul3, mul4] = [mul(2), mul(3), mul(4)]
[ [Function], [Function], [Function] ]
> mul2(3).then((v)=>p(v))
> 6
> p(t()); mul2(3).then((v)=>mul4(v)).
... then((v)=>p(`${t()}: ${v}`))
13:58:11 GMT-0500 (EST)
> 13:58:15 GMT-0500 (EST): 24
> err =
... curriedPromise((a, b) => { throw new Error('err');
})
> err(1)(2).catch((e) => p(`caught ${e}`))
> caught Error: err
```

Promise.all()

Given an **iterable** of promises, returns a promise containing array of fulfilled values, or rejection if any promise rejected.

```
> Promise.all([mul2(3), mul3(4), mul4(5)]).
... then((v) => p(v))
Promise { <pending>, ... }
> [ 6, 12, 20 ]

> Promise.all([mul2(3), err(3)(2), mul3(4), mul4(5)]).
... then((v) => p(v)).
... catch((e) => p(`caught ${e}`))
Promise {  <pending>, ... }
> caught Error: err
```

Promise.all() Continued

`Promise.all()` runs all promises in parallel:

```
> p(t()); Promise.all([mul2(3), mul3(4), mul4(5)]).
... then((v) => p(`${t()}: ${v}`))
15:49:41 GMT-0500 (EST)
Promise { <pending>, ... }
> 15:49:43 GMT-0500 (EST): 6,12,20
```

Took 2 seconds to run all 3 functions even though each function takes 2 seconds apiece.

Promise.race()

Given an **iterable** of promises, returns a promise containing settlement of which ever incoming promise completes first.

```
> Promise.race([mul2(3), mul3(4), mul4(5)]).
... then((v) => p(v))
Promise { <pending>, ... }
> 6

>
```

Static Promise Sequencing

Can sequence promises statically by chaining `then()`'s. Repeating earlier example where output of one async function is input to the next, but using above `mul`*I* functions:

```
> p(t()); Promise.resolve(1).
... then((v) => mul2(v)).
... then((v) => mul3(v)).
... then((v) => mul4(v)).
... then((v) => { p(t()); p(v); })
16:32:12 GMT-0500 (EST)
Promise { <pending>, ... }
> 16:32:18 GMT-0500 (EST)
24

>
```

Dynamic Promise Sequencing

When number of promises to be sequenced is not known statically, we cannot use a static `then()`-chain.

In promise-seq.js

```
//Given promiseFns as an iterable of single argument
//functions returning a Promise.
//Returns promise resulting from sequentially applying
//each function in promiseFns to result of previous
//one, starting with initial promise init.
function sequencePromises(init, promiseFns) {
  let p = init;
  for (const fn of promiseFns) p = p.then((v)=>fn(v));
  return p;
}
```

Dynamic Promise Sequencing Log

```
> .load promise-seq.js
> fns = [mul2, mul3, mul4]
> init = Promise.resolve(1)                //1
> p(t()); sequencePromises(init, fns).  //1*2*3*4
... then((v) => p(`${t()}: ${v}`))
17:15:41 GMT-0500 (EST)
> 17:15:47 GMT-0500 (EST): 24
> fns.push(mul4)  //fns = [mul2, mul3, mul4, mul4]
4
> p(t()); sequencePromises(init, fns).  //1*2*3*4*4
... then((v) => p(`${t()}: ${v}`))
17:16:14 GMT-0500 (EST)
> 17:16:22 GMT-0500 (EST): 96
```

Alternative Dynamic Promise Sequencing

```
//Use reduce when promiseFns is an array
function reduceSequencePromises(init, promiseFns) {
  return promiseFns.reduce(function(acc, fn) {
    return acc.then((v) => fn(v))
  }, init);
}
```

`async` / `await`

- Converts promise code into synchronous style.
- If a function or function expression has the `async` (contextual) keyword in front of it, then that function always returns a promise.
- When the `await` (contextual) keyword is used in front of an expression which is a promise, it blocks the program until the promise is settled. The value of an `await` expression is the fulfillment value of the promise.
- The `await` keyword can only be used within a `async` function.
- Errors can be handled using `try-catch`.
- Seems a big win.

async / await Example

```
> function msgPromise() {
... return new Promise(function (resolve) {
..... setTimeout(() => resolve(`hello@${t()}`),
2000)});
... }
undefined
> async function msg(n) {
... const m = await msgPromise();
... return `${n}: ${m}`
... }
undefined
```

```
> ( async () => {        //must use async to use await
... p(await msg(22));
... p(await msg(42));
... })()                 //async IIFE
Promise { <pending>, ... }
> 22: hello@21:06:53 GMT-0500 (EST)
42: hello@21:06:55 GMT-0500 (EST)

>
```

Async sleep()

```
> async function sleep(millis) {
    return new Promise((resolve) =>
      setTimeout(() => resolve(), millis));
}
> (async function() {
    p(t()); await sleep(2000); p(t()); }
  )()
14:12:38 GMT-0500 (EST)
Promise { <pending>, ... }
> 14:12:40 GMT-0500 (EST)
```

Revisit Dynamic Sequencing using async

In async-seq.js:

```
//Given promiseFns as an iterable of single argument
//async functions returning a Promise.
//Returns promise resulting from sequentially applying
//each function in promiseFns to result of previous
//one, starting with initial promise init.
async function asyncSequencePromises(init, promiseFns)
{
  let v = await init;
  for (const fn of promiseFns) v = await fn(v);
  return v;
}
```

async Dynamic Sequencing Log

```
[amul2, amul3, amul4] =  //destructuring
... [mul2, mul3, mul4].map((m) => async (a) => m(a))
> (async () => p(await amul2(3)))() //IIFE to test amul2
Promise {  <pending>, ... }
> 6
> afns = [amul2, amul3, amul4]
[ [AsyncFunction], [AsyncFunction], [AsyncFunction] ]
> afns.push(async function(a) { return  p(a); })
4
> asyncSequencePromises(init, afns);
Promise { <pending>, ... }
> 24
```