Overview

- Cookbook approach to asynchronous code in modern JavaScript.
- 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`.

Cookbook Approach

Can use asynchronous code without understanding underlying concepts by using "keywords" `async` and `await`:

- If a function is documented as `async` or as returning a promise, then it is possible to call it using the `await` keyword.
- The `await` returns with the success value only when the underlying asynchronous operation completes.
- If an error occurs in the asynchronous function, then the resulting exception can be handled using the usual `try-catch`.
- The `await` keyword can **only** be used within functions declared using the `async` keyword. Consequently, any use of asynchronous code within a program will necessitate declaring the top-level function in the program `async`.
- Enables writing asynchronous code in a synchronous style.

Cookbook Approach to using fetch()

The browser provides fetch() to allow accessing resources asynchronously.

```
async function getUrl(url) {
  const response = await fetch(url);
  if (!response.ok) {
    throw new Error('HTTP error '${response.status}'');
  }
  else {
    return await response.json();
  }
}
```

Cookbook Approach to using MongoDB

Docs for connect(), db, collection, findOne() and close().

```
const mongo = require('mongodb').MongoClient;
async function(mongoUrl, dbName, collectionName, id) {
  //bad code: db connections should usually be reused
  try {
    const client = await mongo.connect(mongoUrl);
    const db = client.db(dbName);
    const collection = db.collection(collectionName);
    const value = await collection.findOne({_id: id});
    await client.close();
    return value;
  }
  catch (err) {
    console.error(err);
    throw err;
  }
```

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.
- Note that browser responsiveness is usually controlled by I/O responsiveness.

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.
- Lower overhead for I/O bound tasks; well suited for browser environment.

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

Try using a global var to get hold of return value.

```
> 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!

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 an 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)`.

Outline of Using Promises for Asynch Operations

```
function doOperation(...params) {
  return new Promise((resolve, reject) => {
    asyncOperation(...params, (result) => { //callback
      if (isOk(result)) {
        resolve(result);
      }
      else {
        reject(result);
      }
    });
  });
}

doOperation(...).
  then(result => { ... }).
  catch( err => ...);
```

Promise States

Pending	The underlying operation is not yet complete.
Settled	The underlying operation completed; it is known whether or not it succeeded resulting in two settled sub-states:

Fulfilled	The underlying operation completed successfully.
Rejected	The underlying operation failed.

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: catch()

```
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

>
```

Promise.all()

Given an **iterable** of promises, returns a promise containing array of fulfilled values, or rejection if any promise rejected. (`mulN(i)` returns promise for `N*i` after 2 second delay):

```
> 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

>
```

A Glimpse at Generators

Generators defined using `function*` and `yield`.

```
> function* seq(lo=0, hi=Number.POSITIVE_INFINITY) {
    for (let i = Math.floor(lo); i <= hi; i++) yield(i);
  }
undefined
> for (s of seq(1, 3)) console.log(s);
1
2
3
undefined
>
```

Generators Return Iterators

- When a generator is called it does not run the generator code, but immediately returns an **iterator**.
- Generator code can `yield` successive values; `return` terminates the generator.
- Caller interacts with returned iterator to step the generator.
- Iterators have a `next()` method which returns an object with two properties:

`done`	A boolean which is true when the generator is done.
`value`	The currently yielded value.

- Passing argument to `next()` makes argument the value returned by `yield`.

Waiting For An Async Value Using Generators

```
function* asyncFn() {
  const value = yield new Promise((resolve) => {
    setTimeout(() => resolve(42), 2000);
  });
  //we can access value
  console.log(value); //outputs 42
}

//drive generator
const iterator = asyncFn();
const iteration = iterator.next();
iteration.value.then(v => iterator.next(v));
```

async / await

- Converts promise code into synchronous style by `yielding` promises from generators.
- 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.
- Note that we may need to fall back on promises using Promise.all() when we want to run code in parallel rather than sequentially.

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 / await Example: Invoking using IIFE

```
> ( 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)
```