Asynchronous Programming in Javascript, Part 2

CSCI 5828: Foundations of Software Engineering
Lectures 30 — 12/10/2015

Goals

- Discuss the notion of asynchronous programming in Javascript in Lecture 29
  - The gap between “now” and “later”
  - The event loop
- Traditional Approach: Callbacks and Callback “hell”
- This lecture => Discuss New Approach: Promises

Callback “Hell”

- A reminder
  - We finished last lecture by discussing the traditional approach to async programming in Javascript
    - Callbacks
  - and the associated problems with them (callback hell)
    - The Pyramid of Doom
    - Unclear evaluation (now and later)
    - Hardcoded Paths (and unclear error handling)
    - Issues of Trust
      - callback too soon? too late? never? in the correct way?
- How can we solve callback hell?

How do we address callback hell?

- With a higher-level abstraction called ***promises***
  - promises were added to JavaScript as part of work on “ES6”
    - which stands for ECMAScript 6
      - which became the official version of JavaScript in June 2015

- ECMAScript 6 brings lots of changes to the language
  - I’ve been using “old school” JavaScript (i.e. ES5) in all of my examples so far; the only ES6 code that I use coming up is promise-related to keep things focused
    - except that I may use ES6 string interpolation here and there
      - var n = 42; var s = `The meaning of life is ${n}.`

Promises (I)

- Promises are
  - an *abstraction* useful in async programming
  - an *associated API* that allows us to use this abstraction in our programs
- A **promise** represents a *future value* of some sort
  - When a promise is created, it is *pending*
    - At some point in the future, the promise is either *fulfilled* or *rejected*
      - *fulfilled* means the promise's computation **succeeded**
      - *rejected* means the promise's computation **failed**
  - Once a promise has either been fulfilled or rejected, it is considered *settled*

Promises (II)

- The promises API has many methods
  - but the most basic interaction will look something like

```
var p = new Promise(
  function(resolve, reject) {
    // long running computation
    if (success) {
      resolve(…);
    } else {
      reject(…);
    }
  }
);
```

A promise wraps code that will run asynchronously

At some point that code is done; if it succeeds, it will notify the world using the resolve callback; if it fails, it will call the reject callback

```
var fulfilled = function(…) {…}; var rejected = function(…) {…};
p.then(fulfilled, rejected);
```

We can register interest in the promise by calling its then() method and providing callbacks for the success/failure cases

Promises (III)

- Since promises help us dealing with asynchronous code, we still have the issue of *now* and *later* to deal with
  - Let's make sure we understand ***when*** promise-related code is executed
- First, consider this code
  - `var p = new Promise( function(resolve, reject) { resolve("now"); } );`
- The code inside of the anonymous function is executed **NOW**
  - that is synchronously
- The code inside of the function will have run to completion by the time the call to "new Promise" returns and stores a value in `p`
  - That code may still take a long time to run; all you need to do is call setTimeout() or some other async function; in this case, the code inside the anonymous function completes **NOW** but the promise will be resolved **LATER**

Promises (IV)

- Second, consider this code
  - `p.then(function(value) { console.log(value); });`
- The callback that is passed to `then()` is executed **LATER**
  - Once the promise has settled (*fulfilled* or *rejected*), all registered callbacks are scheduled in the same way that we saw for `process.nextTick()`
    - That is, when the promise gets resolved or rejected, the callbacks that were registered by calling then() get added to the ***front*** of the event queue
      - that means that they are always executed *asynchronously* **never** *synchronously*
- Let's see this in action

Promises (V)

- Two functions: testNow/testLater
- Both are designed to show when promise-related code is executed
- In testNow, we resolve the promise right away; in testLater we have a call to setTimeout() that delays when the promise gets resolved
- In both cases, we call then() on the returned promise to show when those callbacks run

```
1
2  var testNow = function() {
3    console.log(`testNow  :    1: ${Date.now()}`)
4    var p = new Promise(
5      function(resolve, reject) {
6        console.log(`testNow  :    ?: ${Date.now()}`)
7        resolve(Date.now());
8      }
9    );
10   console.log(`testNow  :    2: ${Date.now()}`)
11   return p;
12 };
13
14 var testLater = function() {
15   console.log(`testLater:    1: ${Date.now()}`)
16   var p = new Promise(
17     function(resolve, reject) {
18       setTimeout(function() {
19         console.log(`testLater:    ?: ${Date.now()}`)
20         resolve(Date.now());
21       }, 0);
22     }
23   );
24   console.log(`testLater:    2: ${Date.now()}`)
25   return p;
26 };
27
28 var p = testNow();
29
30 p.then(function (value) {
31   console.log(`testNow  : then: ${value}`);
32 });
33
34 p = testLater();
35
36 p.then(function (value) {
37   console.log(`testLater: then: ${value}`);
38 });
```

[scheduled.js]

Promises (VI)

- The results?
  - `testNow  :    1: 1449707213272`
  - `testNow  :    ?: 1449707213273` code in "new Promise" is synchronous
  - `testNow  :    2: 1449707213274`
  - `testLater:    1: 1449707213275`
  - `testLater:    2: 1449707213275` — End of main program
  - `testNow  : then: 1449707213274` — *then()* handler put on the **front** of event queue
  - `testLater:    ?: 1449707213276`
  - `testLater: then: 1449707213276` — *setTimeout()* placed its handler at the **end** of the event queue

Once the second promise was fulfilled, the second *then()* handler placed at **front** of event queue

[scheduled.js]

Promises (VII)

- We've seen some differences but, at this point, you may be wondering if promises are any different from callbacks?
  - We shall explore the differences next, but, at a high level,
    - promises are *objects* that represent the result of an async computation
    - once a promise is settled, it stays settled, and remembers its result
    - you can call a promise's then() method more than once and it will ensure that the appropriate callback is always invoked
    - promises can be *chained*, allowing the clean specification of **asynchronous workflows**

Simple Example

- A simple example of using promises to wrap a call to fs.stats() in Node.js

```
1  var fs = require('fs');
2
3  var name = process.argv[2] // get filename from command line
4
5  var p = new Promise(
6    function(resolve, reject) {
7      fs.stat(name, function(err, stats) {
8        if (err) {
9          reject(err);
10         return;
11       }
12       resolve(stats.size);
13     })
14   }
15 );
16
17 var fulfilled = function(size) {
18   console.log(`The size of ${name} is ${size} bytes.`);
19 }
20
21 var rejected = function(err) {
22   console.log(`Unable to determine the size of ${name}.`);
23   console.log(err.message);
24 }
25
26 p.then(fulfilled, rejected)
27
```

[get_size.js]

Note: it doesn't matter how long it takes fs.stat() to do its job

Once it is done, the fulfilled() or rejected() callback is ***guaranteed*** to be called

not too early
not too late
not multiple times
just once, guaranteed!

What if fs.stat() never finishes? We'll talk about that later.

Promise.resolve(): Intro

- Before promises were added to ES6, there were many different promise implementations available
    - each with slightly different APIs and/or semantics
- The ES6 designers were clever; they wanted to come up with a way to convert ANY value to an ES6 Promise, including “promises” from these other libraries
    - after all, a Promise is simply a placeholder for a “future value” (be it a successful result or an error condition)
- The way they did this was to specify a special function called Promise.resolve()
    - Let’s see what it can do

Promise.resolve(): Examples (I)

- `var fulfilled = function(value) { console.log(`Success: ${value}`); }`
- `var rejected = function(err) { console.log(`Error: ${err}`); }`
- `var p = Promise.resolve( 42 );`
- `p.then(fulfilled, rejected); // Prints: “Success: 42”`

- If you pass in a value to Promise.resolve, it creates a promise that has been fulfilled with that value. As you can see, when we call then() on p, it passes 42 as the value of the promise, just as if we had done the following:

- `var p = new Promise(function(resolve, reject) { resolve(42) });`

- Promise.resolve() will do this for ***any*** value, including collections and null
  - Side note: Promise.reject() does the opposite, taking a value and creating a rejected promise with that value

- `var p = Promise.reject( 42 );`
- `p.then(fulfilled, rejected); // Prints: “Error: 42”`

`[resolve1.js]`

Promise.resolve(): Examples (II)

```
• var fulfilled = function(value) { console.log(`Success: ${value}`); }
• var rejected = function(err) { console.log(`Error: ${err}`); }
• var p1 = Promise.resolve( 42 );
• var p2 = Promise.resolve( p1 );
• console.log(`p1 and p2 are the same object: ${p1 === p2}`);
```

- In the code above, we show what happens when we pass a promise to Promise.resolve().
  - We start by creating p1 and then we pass p1 into Promise.resolve() to create p2.
    - We then use the === operator to determine if p1 and p2 are identical (the same object) and it returns true
      - We can show that === tests identity by trying the line below
        - `{name: "ken"} === {name: “ken"} // evaluates to false`

[resolve2.js]

Promise.resolve(): Examples (III)

- `var fulfilled = function(value) { console.log(`Success: ${value}`); }`
- `var rejected = function(err) { console.log(`Error: ${err}`); }`
- `var successObj = { then: function(cb) { cb(42); }};`
- `var failObj = { then: function(cb,err) { err("ouch"); }};`
- `var p1 = Promise.resolve( successObj );`
- `var p2 = Promise.resolve( failObj );`
- `p1.then(fulfilled, rejected); // Prints: Success: 42`
- `p2.then(fulfilled, rejected); // Prints: Error: ouch`

- This code shows what happens if you pass in an object to Promise.resolve() that has a `then()` method on it; the ES6 designers decided that the way to identify other types of "promise" objects was to see if they have a `then()` method and call it! They call this a "*thenable object*".
- Duck typing: if it ***looks*** like a promise, and ***acts*** like a promise, it must ***be*** a promise!
- If the object's `then()` method calls the first function passed to it, the new promise is fulfilled; if it calls the second function passed to it, rejected.
  - In this way, other types of promises can be converted to ES6 promises

`[resolve3.js]`

Promises: Passing Values

- As mentioned above, promises represent a "future value"
  - You can pass one and only one value to then() callbacks
    - `var p = Promise.resolve( 42 );`
    - `p.then(function(value) { // value == 42 });`
  - As you can see, a callback function in then() receives one value
    - what I called `fulfilled` and `rejected` in the slides above
  - There's no way to pass multiple parameters to these functions
    - you either get a value or an error condition
  - If you need to pass multiple values, you need to wrap them in an object or an array; e.g. [23, 42]

`[passing1.js]`

Promises: Flow Control

- However, you don't **have** to use the value that is passed
  - You also don't have to **pass** a value; if you don't, **null** is passed instead
- In this case, you're just using a promise as a way to do asynchronous workflows, where one step in the process needs to know when a previous step is finished

```
var step1 = new Promise(
  function(resolve, reject) {
    setTimeout(function() {
      resolve(); // call resolve without passing a value
    }, 5000);
  }
);
step1.then(function() { console.log("Step 1 is done!"); });
```

- Here, it takes 5 seconds before our function calls resolve(). Only then, does our callback get notified

[passing2.js]

Promises can be chained

- How can we use promises to specify an asynchronous workflow?
  - We need some way to be able to specify the steps of that workflow
- In our examples so far, we've only seen single step workflows; create a promise and call `then()` on it
- But, promises can be **chained** together; now things get interesting!

- To make this work, you need to know two things
  - when you call `then(fulfilled, rejected)`, it returns a new promise!
    - We've been ignoring that promise so far… no longer!
  - that new promise can either be fulfilled or rejected
    - it is rejected if the original promise was rejected
    - it is fulfilled if we ***return a value*** from the `fulfilled()` callback!

Creating the Chain (I)

- Start with a simple promise
  - `var step1 = Promise.resolve( 21 );`
- Create a function that performs the action of step2
  - `var double = function(v) { return v * 2; };`
- Create step2 by calling then() on step1
  - `var step2 = step1.then(double, rejected);`
- step2 is a promise, because
  - `then()` returns a newly created promise (as we said on the previous slide)
  - it's value is the value returned by the "fulfillment callback", `double()`
- We can now call `then()` on step2!
  - `step2.then(fulfilled, rejected) // prints "Success: 42"`

`[chain1.js]`

Creating the Chain (II)

- It's hard to "see the chain" on the previous slide, let's make it more clear
  - `Promise.resolve( 21 ).then(double, rejected).then(fulfilled, rejected)`
- There it is!
  - We don't have to store each promise returned by `then()`, we can just immediately call `then()` on them
    - It doesn't matter how long each step takes
      - If one of the promises has a function that takes a long time to compute, then the workflow pauses until it's ready
        - it then calls resolve or returns a value, which fulfills the promise, which triggers a call on the next registered callback

`[chain1.js]`

Creating the Chain (III)

```
function delay(time) {
  return new Promise(
    function(resolve,reject){
      setTimeout( resolve, time );
    }
  );
}

delay( 1000 ).then(
  function STEP2() {
    console.log( "step 2 (after 1000ms)" );
    return delay( 500 );
  }
).then(
  function STEP3() {
    console.log( "step 3 (after another 500ms)" );
  }
).then(
  function STEP4() {
    console.log( "step 4 (next Job)" );
    return delay( 300 );
  }
).then(
  function STEP5() {
    console.log( "step 5 (after another 300ms)" );
  }
);

```

[chain2.js]

- Chains can be as long as we want and each step can take as long as it needs
  - Here the `delay()` function returns a promise that uses `setTimeout()` to delay for a specified period of time
  - In each step (except step 3), it uses `delay()` to return a promise that will eventually trigger the next call to `then()`
  - When you return a promise from `then()`'s fulfillment callback, that promise is used as `then()`'s return value

Error Handling (I)

- Javascript has a try/catch block for handling exceptions

```
try {
  throw Error("Whoops!");
} catch (err) {
  console.log(`${err}`);
}
```

- But, it only works for synchronous code, not asynchronous

```
try {
  setTimeout(function() {throw Error("Whoops!"); }, 0);
} catch (err) {
  console.log(`${err}`);
}
console.log("Where is my error?");
```

[error1.js and error2.js]

Error Handling (II)

- Handling errors with callbacks is possible but fraught with peril
  - It is hard to compose error handling across a chain of callbacks
  - And, you typically, have to put in a lot of if statements to handle the conditional logic, leading you back to *callback hell*
- Nevertheless, you will see conventions such as Node.js's error first approach
  - `fs.stat("file.txt", function (err, result) { … });`
- As mentioned in the last lecture, the variable "err" will be set to null if the call to fs.stat was successful. If it is not null, then it is likely an instance of an exception and you need to handle it. As we saw last time,
  - `if (err) throw err;`
- is a common way to (not) handle errors passed in this way

Error Handling (III)

- Promises provide a clean way to handle asynchronous errors
  - We've already seen the mechanism, we just haven't seen an example of it
- If you encounter an error while trying to resolve a promise, you handle the error by catching it and passing it to reject().
- This set's the state of the promise to rejected, which then becomes its immutable state for the rest of the program
  - If you call `then()` on a rejected promise, your rejected callback will be invoked
  - You can also call `catch()` on a promise; it takes just one callback which acts like `then()`'s rejected callback; just like `then()`, `catch()` returns a promise that can also be chained.

Error Handling (IV)

- These two simple examples demonstrate what happens when you register a callback on a rejected promise
- `Promise.reject("whoops").then(null, function(err) { console.log(err); });`
- `Promise.reject("whoops").catch(function(err) { console.log(err); });`
    - Here, we called `Promise.reject()` directly to create a rejected promise but promises take care of other error cases; consider:
- `var p = new Promise( function(resolve, reject) { foo.bar(); } );`
    - We never called `resolve()` or `reject()`, what's the state of the promise?
    - The code inside the function references an object (`foo`) that doesn't exist; Javascript throws a ReferenceError and the promise is going to "catch" that error and become rejected automatically
        - this also happens when an error occurs in a `then()` callback

`[error3.js]`

Error Handling (V)

- Here's an example of an error occurring in a `then()` callback
    - The promise that `then()` returns is automatically rejected and can then be `catch()`-ed.

```
var lower = function(v) { return v.toLowerCase(); };
var rejected = function(err) { console.log(err); };
Promise.resolve(42).then(lower).catch(rejected);
```

- This set's up an important point
    - The error occurred in lower(), producing a promise, and then we could catch() it. If you don't retrieve the promise produced by then() and call catch() on it (or then()), you won't see the error!
        - See example next slide

[error4.js]

Error Handling (VI)

```
var fulfilled = function(value) { console.log(`Success: ${value}`); }
var rejected  = function(err) { console.log(`Error: ${err}`); }

var trouble = function( v ) {
  foo.bar();
  console.log( "The meaning of life: " + v );
}

Promise.resolve(42).then(trouble, rejected)

console.log("Where did the error go?")

Promise.resolve(42).then(trouble, rejected).then(fulfilled, rejected);

console.log("Oh, there it is!")
```

Error gets lost

Error generated and caught
in the next step of the chain

[error5.js]

Error Handling (VII)

- What's nice about the Promise approach to handling errors is that it is possible to perform error recovery
    - If you return a value from an error handler, then that resolves the new promise being created by either then() or catch() and that resolved promise can then be chained.
- Let's look at an example: error_in_chains.js

What's left?

- We've seen
  - `new Promise()`
  - `Promise.resolve()`
  - `Promise.reject()`
  - `p.then();`
  - `p.catch();`
- What's left?
  - `Promise.all()`
  - `Promise.race()`
- The final two methods of the Promise API are `all()` and `race()`

Promise.all()

- `Promise.all()` takes an array of promises and returns a new promise
  - If ***all*** of the input promises **resolve**, then the new promise resolves
  - If ***one*** of the input promises **rejects**, then the new promise rejects
- Let's return to our example that used callbacks to determine the size of all files in a given directory
  - The solution with promises is a little longer but the specification of the asynchronous workflow is MUCH cleaner; it looks like this
    - `get_files('.').then(filter).then(gather).then(sum).then(report).catch(fail);`
- Each step happens asynchronously but it is guaranteed to either report the total size or print out an error message; it makes use of `Promise.all()` twice!

`[file_total.js]`

Promise.race()

- Promise.race() takes an array of promises and returns a new promise
  - The first promise to resolve has its value fulfill the new promise
  - The first promise to reject causes the new promise to reject with its reason
- i.e. it's a race! The first promise to do something (resolve or reject) wins!

- I have two examples to demonstrate the use of Promise.race();
  - race.js: a simple demonstration of the API call
  - fastest_size.js: a more complicated example where we calculate the sizes of all the files in the directory but only report the result for the fastest `fs.stat()` call!

Introduction Complete!

- With that, we have been introduced to the new approach to handling asynchronous events in JavaScript.
- Where might you encounter the use of promises? Everywhere!
- Promises are being used to handle asynchronous events in
  - server side frameworks like node and its vast array of modules
  - web frameworks like React and Angular
    - consider a common workflow
      - receive request to update user;
      - retrieve user from database
      - update user with new info
      - persist user back to database
      - send back HTML that shows updated user (redirect to /user/:id)
      - BUT, if any of these steps fail, show error page (redirect to /error)
    - Promises: get(user).then(update).then(persist).then(show).catch(error);

Summary

- We built on our introduction to asynchronous Javascript by covering promises
  - promises address many of the problems of callback hell
    - no more pyramid of doom
    - well-known invocation semantics
    - asynchronous workflows with well-defined error handling

- Semester Wrap-Up
  - Thank you
  - Looking forward to grading your final presentations; due 11:59 PM on Sunday