305CDE Lab 5

JavaScript Promises

October 2014

Overview

- Callback Hell
- “Future-Facing” Objects: Promises

Callback Hell

Callbacks

Remember that functions are *first class objects* in JS. So they can be used as parameter values ("callbacks") for input to other functions.

```
function some_function(arg1, arg2, callback) {
  var my_number = Math.ceil(Math.random() *
    (arg1 - arg2) + arg2);
  // more code here - may take a while
  callback(my_number);
}
some_function(5, 15, function(num) {
  console.log("callback called! " + num);
});
```

- Third input to `some_function()` is a callback function

Nested Callbacks

But what if `some_function()` itself uses a callback as one of its arguments?

- We get a *nested callback*
- Unfortunately this pattern can continue for several layers

```
some_function(param, function(err, res) {
  some_function2(param, function(err, res) {
    some_function3(param, function(err, res) {
      some_function4(param, function(err, res) {
        some_function5(param, function(err, res) {
          some_function6(param, function(err, res) {
            // do something useful
          });
        });
      });
    });
  });
});
```

The "Problem"

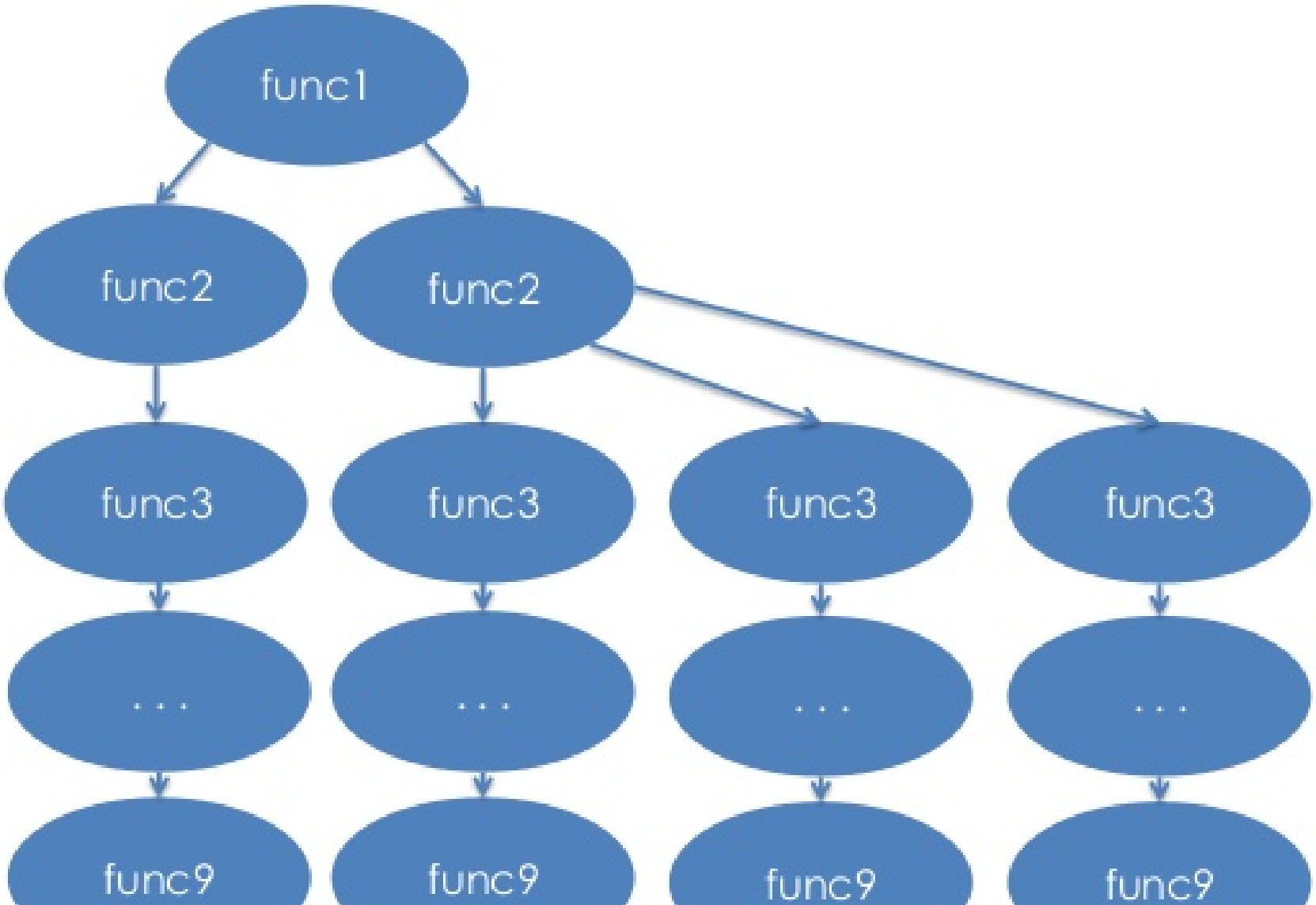
- JS is an event-based language
- So even in moderately complex programs various *chains of events* need to be handled
- Using callbacks to do this (with nesting) makes the code very hard to read
  - Code that is hard to read is hard to debug
  - Code that is hard to read is hard to refactor
  - Code that is hard to read is hard to collaborate on with colleagues

Best case: the function calls are *linear* down the chain - i.e. each function has at most one callback.

$$func_1 \rightarrow func_2 \rightarrow func_3 \rightarrow \ldots \rightarrow func_n$$

The Real Problem

Common case: the function calls are *branched* down the chain - i.e. at least one function has two or more callbacks.



Branching Callbacks

Real problems.

- If the callbacks branch, we can't know the order of the function calls
- It can be very complex to "reassemble" data returned from the various branches correctly
- Scoping becomes a challenge

Possible solution?

- Avoid *anonymous* function callbacks. Replace them with (un-nested) named functions defined in their own blocks. But:
  - It is still almost impossible to quickly infer the "meaning" or "intention" of the code
  - Branching and scoping are still challenges
- Use event listeners when possible

```
var img1 = document.querySelector('.img-1');
img1.addEventListener('load', function() {
  // woo yey image loaded
});
img1.addEventListener('error', function() {
  // argh everything's broken
});
```

- But what about events that happen before binding?
- What about *combinations of events* happening??

A Better Solution

- Use callbacks in very simple (one or two nested layers) situations
- Use listeners for events that can happen multiple times on the same object:
  - `keyup`
  - `click`
  - etc.
- BUT otherwise use *JavaScript Promises* to handle multiple asynchronous event chains
  - in particular success/failure chains arising in AJAX calls!

JavaScript Promises

What Is A JS Promise?

The core idea behind promises is that a **promise object** represents the result of an asynchronous operation. So a promise can be in one of three different states:

1. pending - The initial state of a promise.
2. fulfilled / resolved - The state of a promise representing a successful operation.
3. rejected - The state of a promise representing a failed operation.

Together, the last two are also referred to as *settled*. Once a promise is fulfilled or rejected, it can never change again.

Making Promises

To create a promise object when you need to deal with some async chaining, you just construct a `new Promise()` object:

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...
  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

Note that you specify *under what conditions* to `resolve` or `reject` the promise object.

The `Promise` Constructor

How does the constructor work?

- Takes one argument - a callback function!!!
- The argument has two *function* parameters: `resolve` and `reject`

Inside the callback:

1. Run your async code
2. If it works, invoke `resolve()`
3. If it fails, invoke `reject()`

The latter two invocations pass their values "down the promise chain" to be used later...