Avoiding Callback Hell using Promises in Node.js

Node.js is event-based..

In a normal process cycle the webserver while processing the request will have to wait for the IO operations and thus blocking the next request to be processed.

Node.JS process each request as events, The server doesn’t wait for the IO operation to complete while it can handle other request at the same time.

When the IO operation of first request is completed it will call-back the server to complete the request.

Threads VS Event-driven / Non-Blocking? Blocking?

- By introducing callbacks. Node can move on to other requests and whenever the callback is called, node will process is..
- Non-blocking code is to be read as « put function and params in queue and fire callback when you reach the end of the queue »
- Blocking= return,
Non-Blocking= no return. Only callbacks

Callback Example

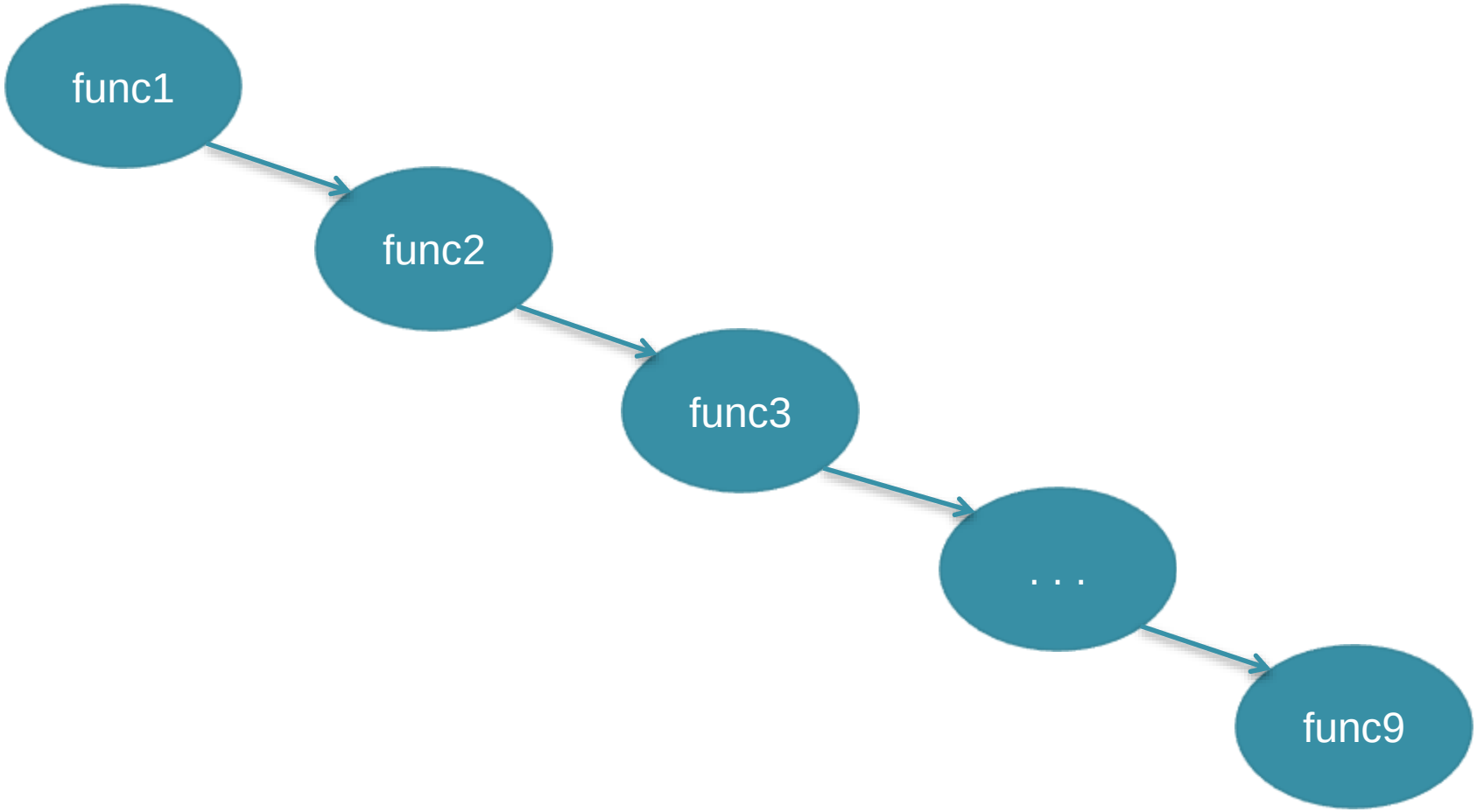
```
fs = require('fs');

fs.readFile('f1.txt','utf8',function(err,data){
    if (err) {
        return console.log(err);
    }
    console.log(data);
});
```

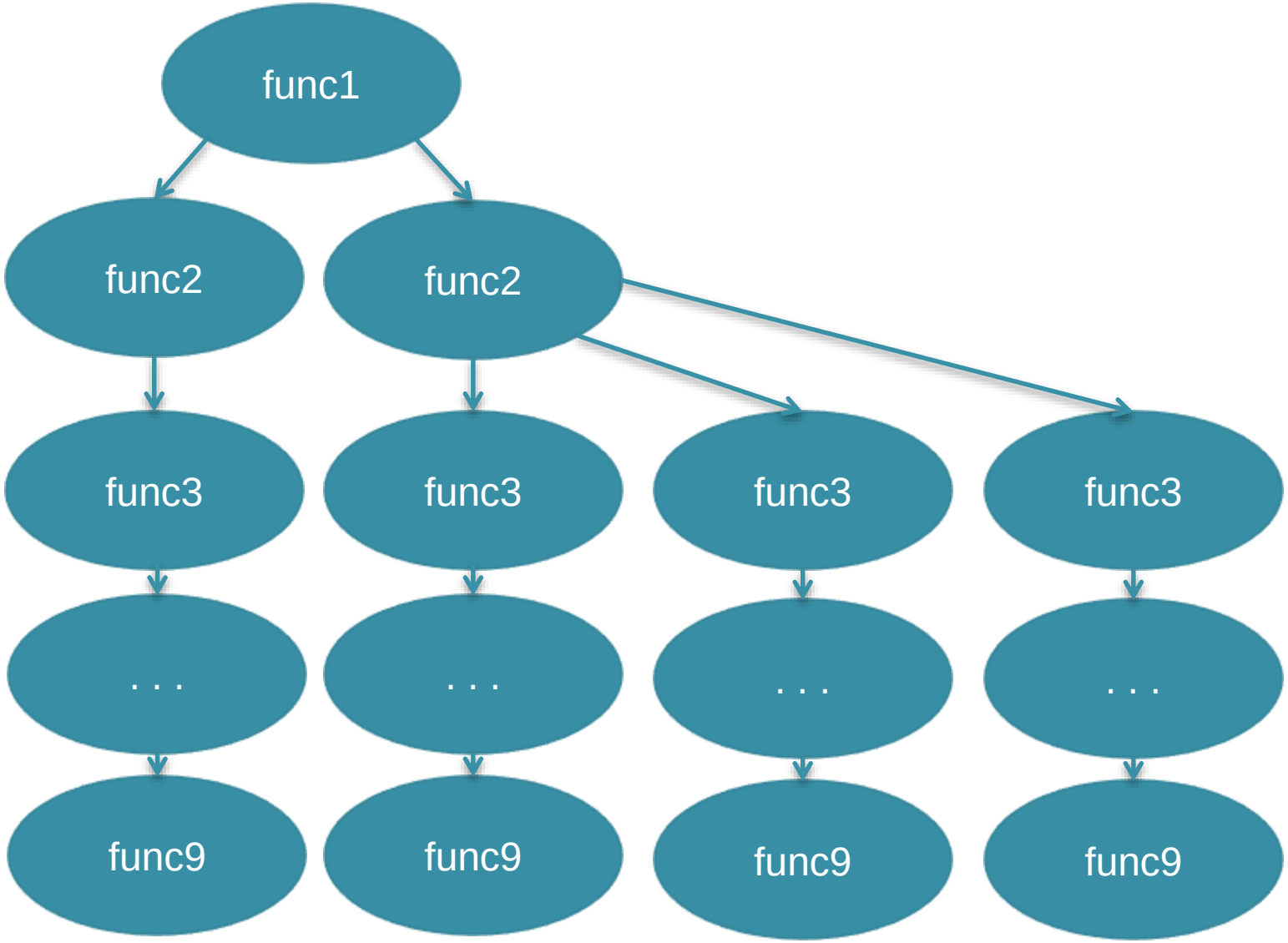
Callback Hell - Pyramid of Doom

```
func1(param, function(err, res) {
  func2(param, function(err, res) {
    func3(param, function(err, res) {
      func4(param, function(err, res) {
        func5(param, function(err, res) {
          func6(param, function(err, res) {
            func7(param, function(err, res) {
              func8(param, function(err, res) {
                func9(param, function(err, res) {
        // Do something…
          });
        });
      });
    });
        });
  });
  });
});
```

Best case, this is linear



But it can branch



Separate Callback

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

callback = function(err,data){
   if (err) {
      return console.log(err);
   }
   console.log(data);
}

fs.readFile('f1.txt','utf8',callback);
```

Can turn this:

```
var db = require('somedatabaseprovider');


 http.get('/recentposts', function(req, res){
    db.openConnection('host', creds, function(err,
                                          conn){
       res.param['posts'].forEach(post) {
          conn.query('select * from users where id=' +
                              post['user'],function(err,results){
             conn.close();
             res.send(results[0]);
          });
       }
    });
});
```

…into this

```
var db = require('somedatabaseprovider');

http.get('/recentposts', afterRecentPosts);

function afterRecentPosts(req, res) {
   db.openConnection('host', creds, function(err, conn) {
     afterDBConnected(res, conn);
   });
}
function afterDBConnected(err, conn) {
   res.param['posts'].forEach(post) {
     conn.query('select * from users where id='+post['user'],afterQuery);
   }
}
function afterQuery(err, results) {
   conn.close();
   res.send(results[0]);
}
```

Good start!

- Callback function separation is a nice aesthetic fix
- The code is more readable, and thus more maintainable
- But it doesn’t improve your control flow

Promises - Description

- Promises take a call to an asynchronous function and wrap it with an object whose methods proxy when the wrapped function either completes or errors
- A good Promise library also provides a set of control methods on that object wrapper to handle composition of multiple Promise-ified asynchronous method calls
- Promises use the best qualities of an object--encapsulation of state--to track the state of an asynchronous call

Again, Why Promises?

- It's a spec: http://wiki.commonjs.org/wiki/Promises/A
- Generally supported by a bunch of libs both browser and server-side:
  - jQuery (sort of, supposedly doesn't fully work like Promises/A)
  - AngularJS
  - Q library (https://github.com/kriskowal/q)
- Provides separation of concerns between wrapping and flow of control handling of deferred activities

Pyramid of Doom Again

```
step1(function (value1) {
    step2(value1, function(value2){
        step3(value2,
    function(value3){
            step4(value3,
                function(value4){
      // Do something with value4
            });
        });
    });
});
```

```
step1 //  a Promise obj
.then(step2)
.then(step3)
.then(step4)
.then(function (value4)
//Do something with value 4
})
.fail( function (error) {
```

- Handle any error from step through step4

```
})
```

Chaining

- ```
  return getUsername()
  .then(function (username) {
      return getUser(username)
      .then(function (user) {
          // if we get here without an error,
          // the value returned here
          // or the exception thrown here
          // resolves the promise returned
          // by the first line
      })
  });
  ```

- ```
  return getUsername()
  .then(function (username) {
      return getUser(username);
  })
  .then(function (user) {
      // if we get here without an error,
      // the value returned here
      // or the exception thrown here
      // resolves the promise returned
      // by the first line
  });
  ```

Nesting

- It’s useful to nest handlers if you need to capture multiple input values in your closure.
- function authenticate()

```
{
   return getUsername()
   .then(function (username) {
            return getUser(username);
    })
    // chained because we will not need the user name in the next event
    .then(function (user) {
          return getPassword() // nested because we need both user and password next
          .then(function (password) {
                   if (user.passwordHash !== hash(password)) {
                        throw new Error("Can't authenticate");
                                 }
          });
    });
}
```

Combination

- return Q.all([
    eventualAdd(2, 2),
    eventualAdd(10, 20)
  ])
  .then (function getAll(…args){
      var firstResult = args[0][0];
      // and so on….
  });
- function eventualAdd(a, b) {
    return Q.spread([a, b], function (a, b) {
      return a + b;
    })
  }

Using Combination and spread

- Using both we can avoid chaining:
  - return getUsername()

```
.then(function (username) {
    return [username, getUser(username)];
})
.spread(function (username, user) {

});
```

Creating Promises - Using Deferreds

- ```
  var deferred = Q.defer();
  FS.readFile("foo.txt", "utf-8", function (error, text) {
      if (error) {
          deferred.reject(new Error(error));
      } else {
          deferred.resolve(text);
      }
  });
  return deferred.promise;
  ```
- ```
  deferred.reject(new Error(error)); // is shorthand for:
  var rejection = Q.fcall(function (error) {
      throw new Error(error);
  });
  deferred.resolve(rejection);
  ```

Some Important Methods of Q Library

- promise.finally(callback)
  - useful for collecting resources regardless of whether a job succeeded, like closing a database connection, shutting a server down, or deleting an unneeded key from an object.
- promise.done(onFulfilled, onRejected, onProgress)
  - This method should be used to terminate chains of promises that will not be passed elsewhere. Since exceptions thrown in then callbacks are consumed and transformed into rejections, exceptions at the end of the chain are easy to accidentally, silently ignore.
  - The Golden Rule of done vs. then usage is: either return your promise to someone else, or if the chain ends with you, call done to terminate it. Terminating with catch is not sufficient because the catch handler may itself throw an error.

Some more…

- promise.delay(ms)
  - If the static version of Q.delay is passed only a single argument, it returns a promise that will be fulfilled with undefined after at least ms milliseconds have passed. (If it's called with two arguments, it uses the usual static-counterpart translation, i.e. Q.delay(value, ms) is equivalent to Q(value).delay(ms).)

Thank You