Intro to Javascript Promises

Mark Zhang

6.148 MIT Web Programming Competition 2017

KENSHO

KENSHO

Technology that brings transparency to complex systems

Strategic Investors

Google ventures

Goldman Sachs

J.P.Morgan

Bank of America Merrill Lynch

Morgan Stanley

citi

STANDARD & POOR'S

CNBC

IQT IN·Q·TEL

Come work with us!

- 70 person start-up
- Offices in Harvard Square, World Trade Center in NYC
- Looking for talented web developers!
- E-mail: jobs@kensho.com

Outline

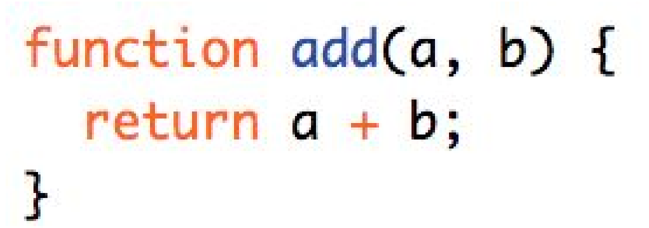
- What are Promises?
- Promise API deep dive
- Why are Promises cool?
- Further topics to explore

What are Promises?

Promises are a way of dealing with eventual values.

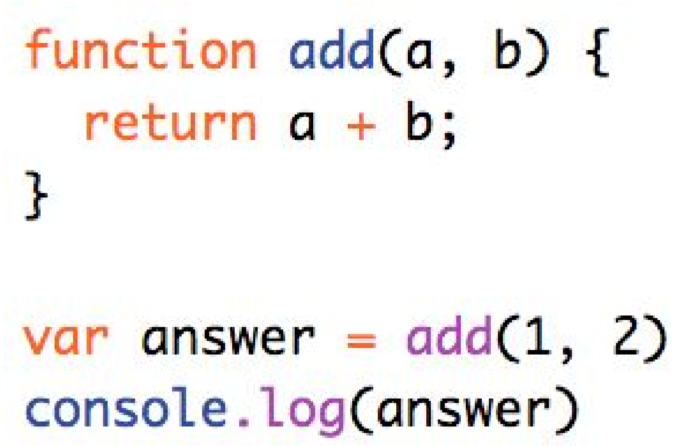
Immediate vs Eventual Values

Immediate Values



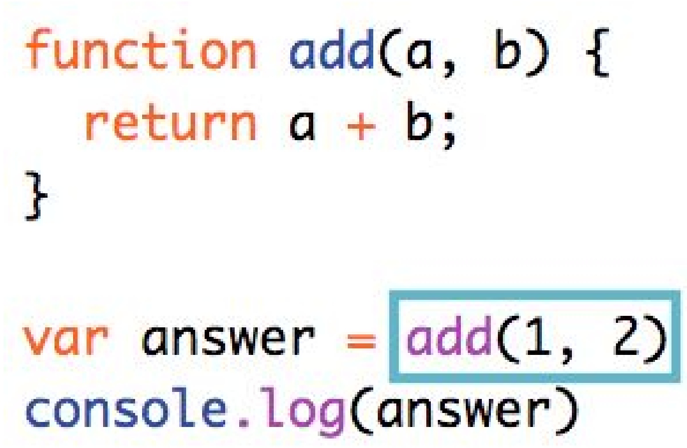
```
function add(a, b) {
  return a + b;
}
```

Immediate Values



```
function add(a, b) {
  return a + b;
}

var answer = add(1, 2)
console.log(answer)
```

Immediate Values

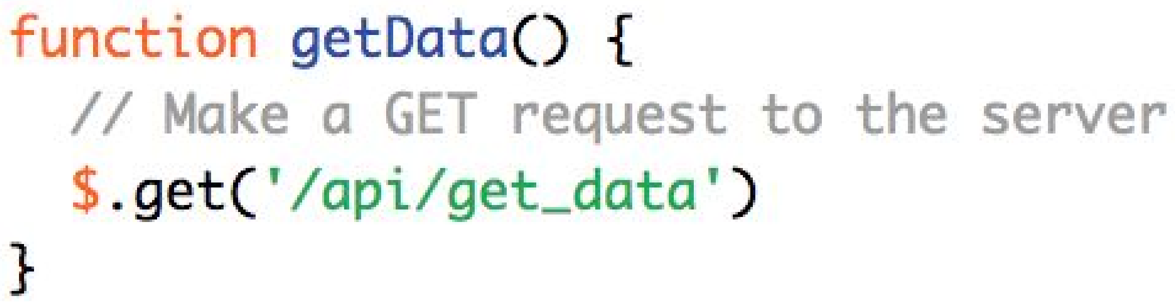


```
function add(a, b) {
  return a + b;
}

var answer = add(1, 2)
console.log(answer)
```

Value immediately available

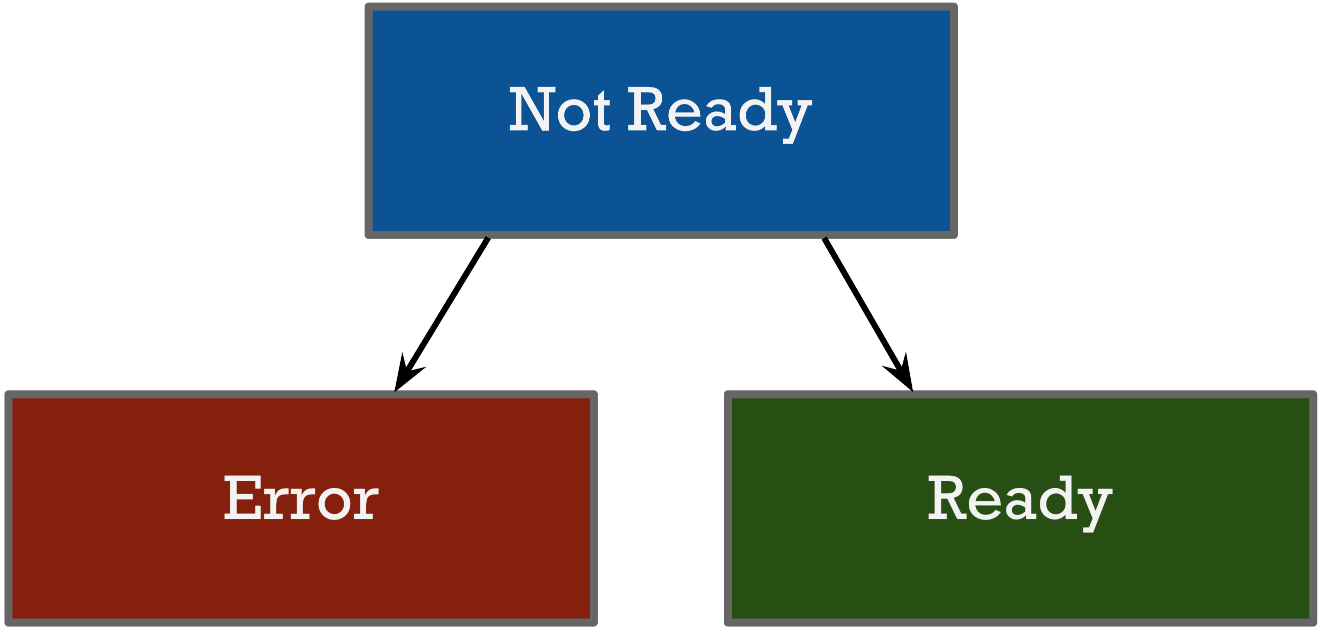
Eventual values aren’t available immediately.

Eventual Values

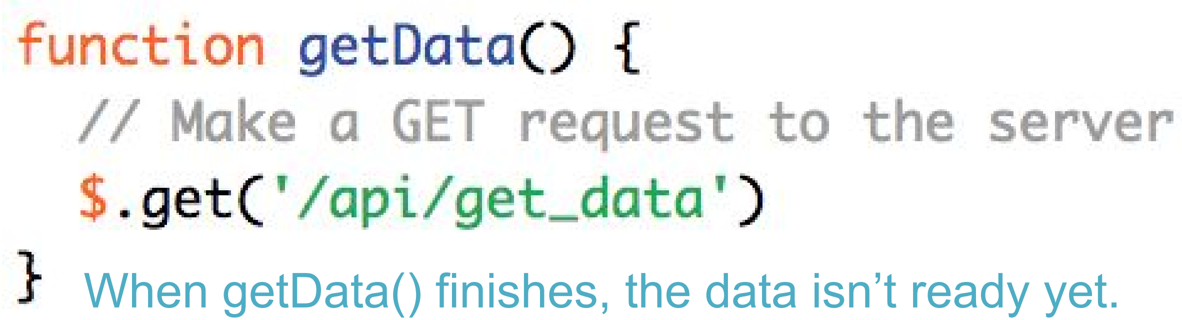


```
function getData() {
  // Make a GET request to the server
  $.get('/api/get_data')
}
```

Eventual Value (State Machine)



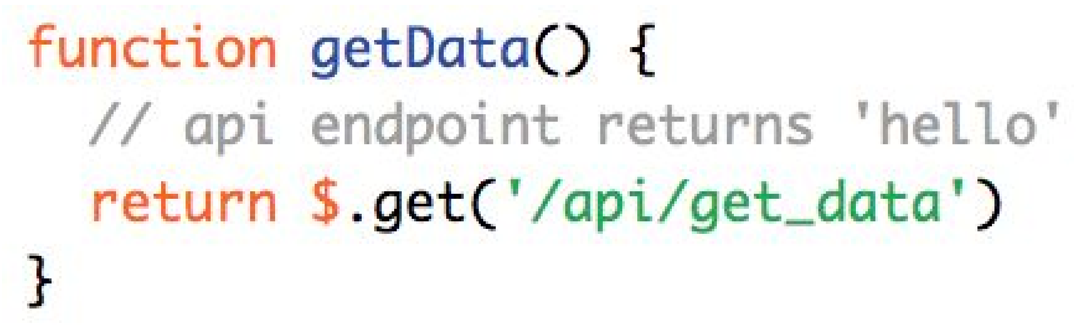
Eventual Values



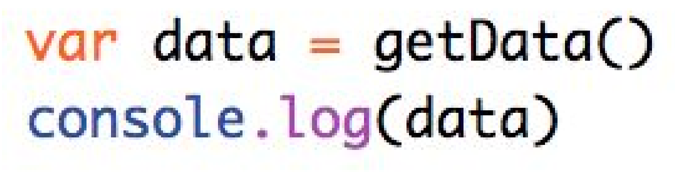
```
function getData() {
  // Make a GET request to the server
  $.get('/api/get_data')
}
```

When getData() finishes, the data isn’t ready yet.

Eventual Values



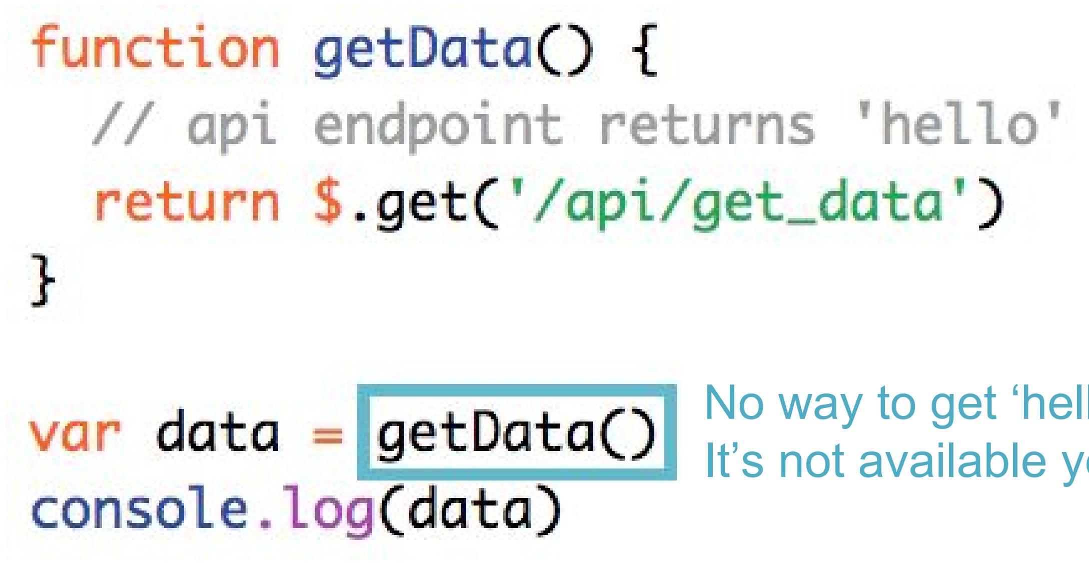
```
function getData() {
  // api endpoint returns 'hello'
  return $.get('/api/get_data')
}
```



```
var data = getData()
console.log(data)
```

DOESN'T WORK!!!

Eventual Values



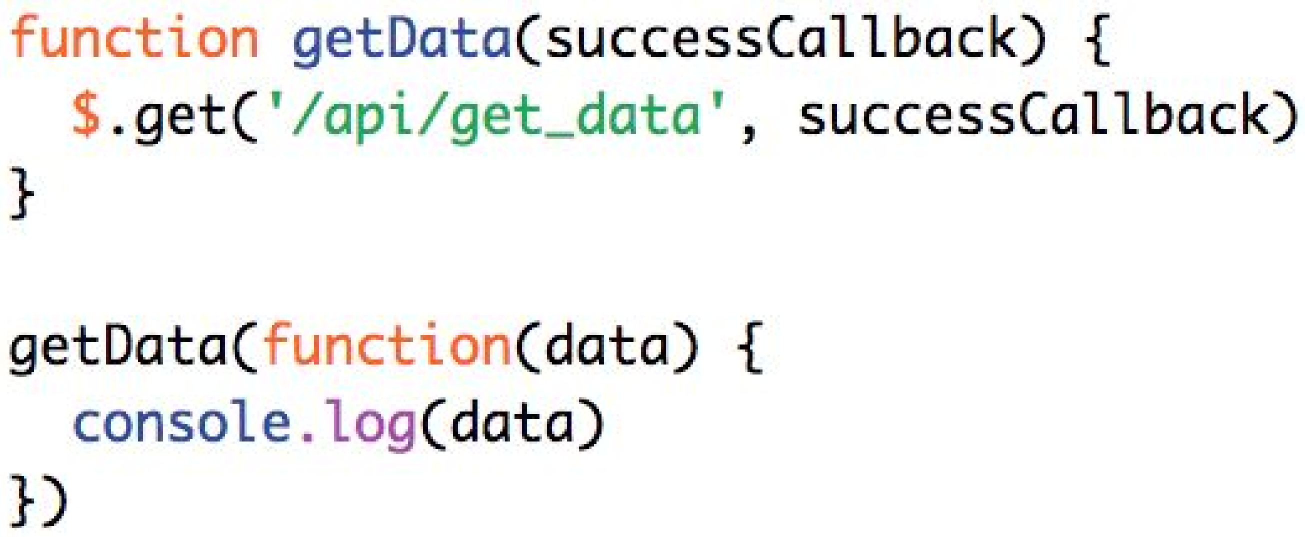
```
function getData() {
  // api endpoint returns 'hello'
  return $.get('/api/get_data')
}

var data = getData()
console.log(data)
```

No way to get ‘hello’ here.
It’s not available yet.

We need some way to handle eventual values once they're ready.

One approach is to assign a callback function to handle the value when it’s ready.

Callbacks



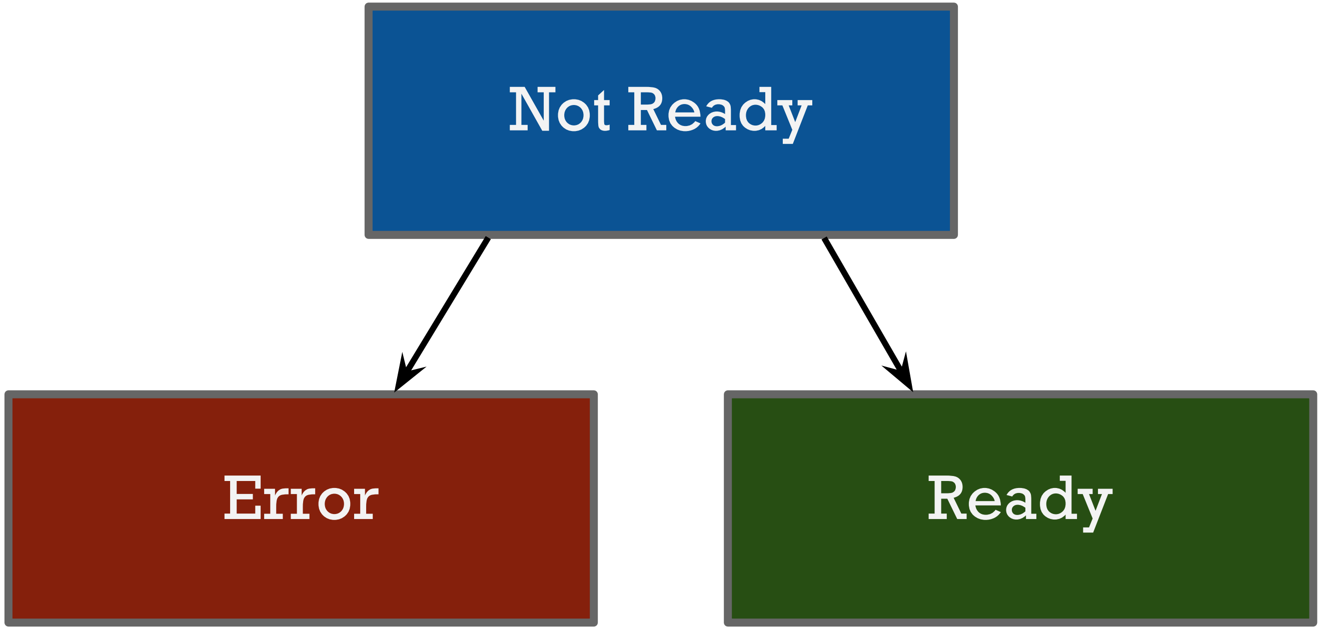
```
function getData(successCallback) {
  $.get('/api/get_data', successCallback)
}

getData(function(data) {
  console.log(data)
})
```

This is known as
continuation-passing style (CPS).

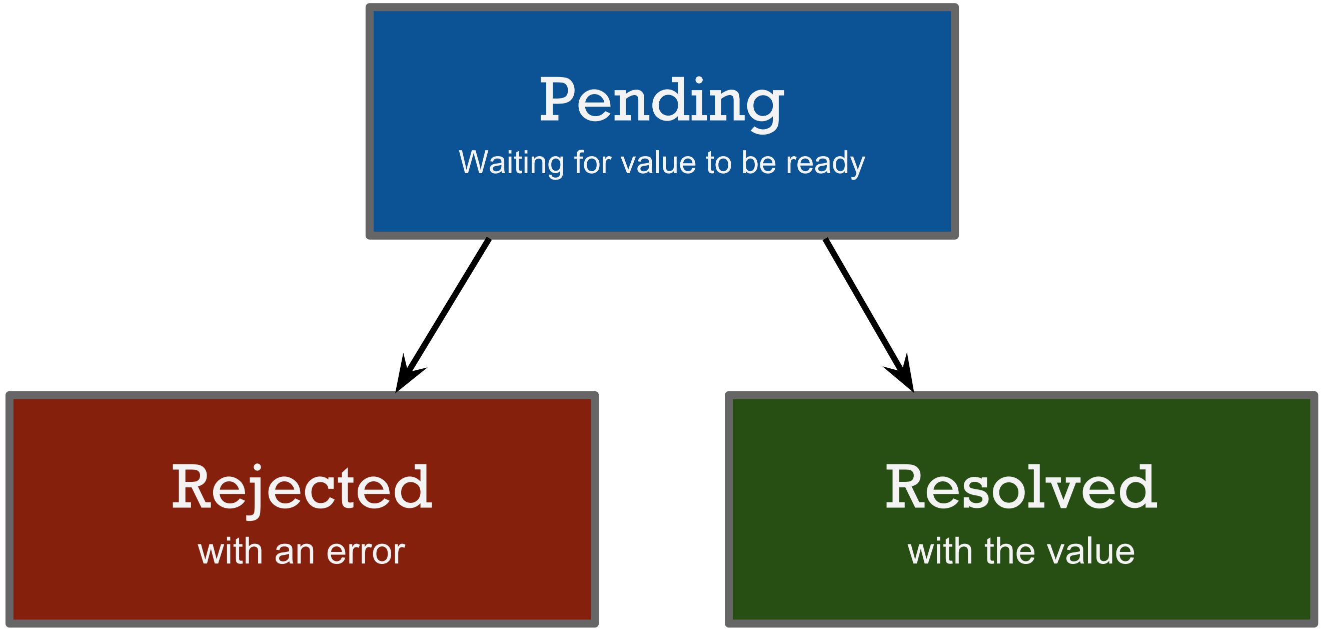
A better approach is to use Promises.

Promises capture the notion of an eventual value in an object.

Eventual Value (State Machine)

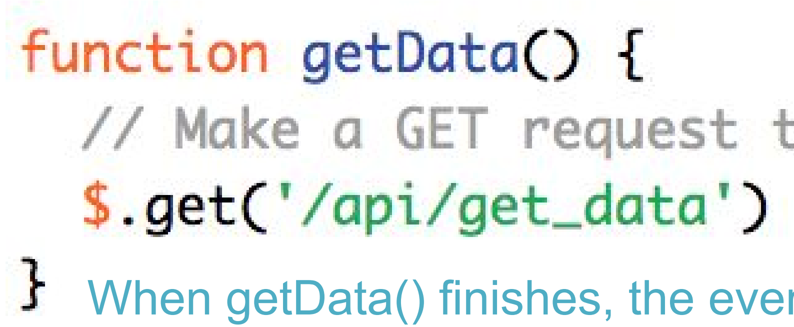


Promise (State Machine)



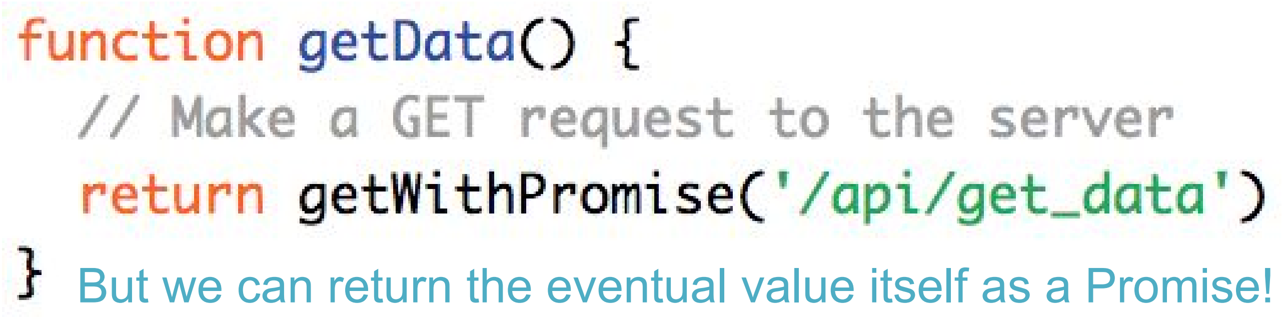
CPS Callbacks handle eventual values.

Promises ARE eventual values.



```
function getData() {
  // Make a GET request to the server
  $.get('/api/get_data')
}
```

When getData() finishes, the eventual value isn’t ready yet.



```
function getData() {
  // Make a GET request to the server
  return getWithPromise('/api/get_data')
}
```

But we can return the eventual value itself as a Promise!

```
function getData() {
  // Make a GET request to the server
  return getWithPromise('/api/get_data')
}

var promise = getData()

promise.then(function(data) {
  console.log(data)
})
```

Promises

```
function getData() {
  return getWithPromise('/api/get_data')
}
```

```
var promise = getData()
```

```
promise.then(function(data) {
  console.log(data)
})
```

Attach a handler to the Promise.

Promises

```
function getData() {
  return getWithPromise('/api/get_data')
}
```

```
var promise = getData()
```

```
promise.then(function(data) {
  console.log(data)
})
```

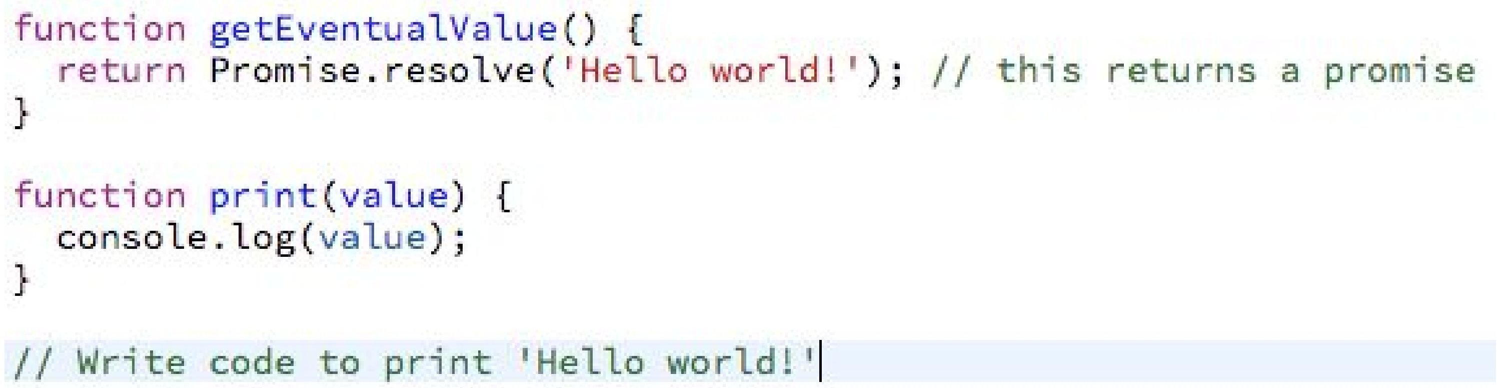
This is run when Promise is resolved.

So far…

- Eventual values are not available immediately.
- CPS uses callback functions to handle eventual values when they are ready.
- Promises act as a proxy for eventual values themselves.

Exercise 1

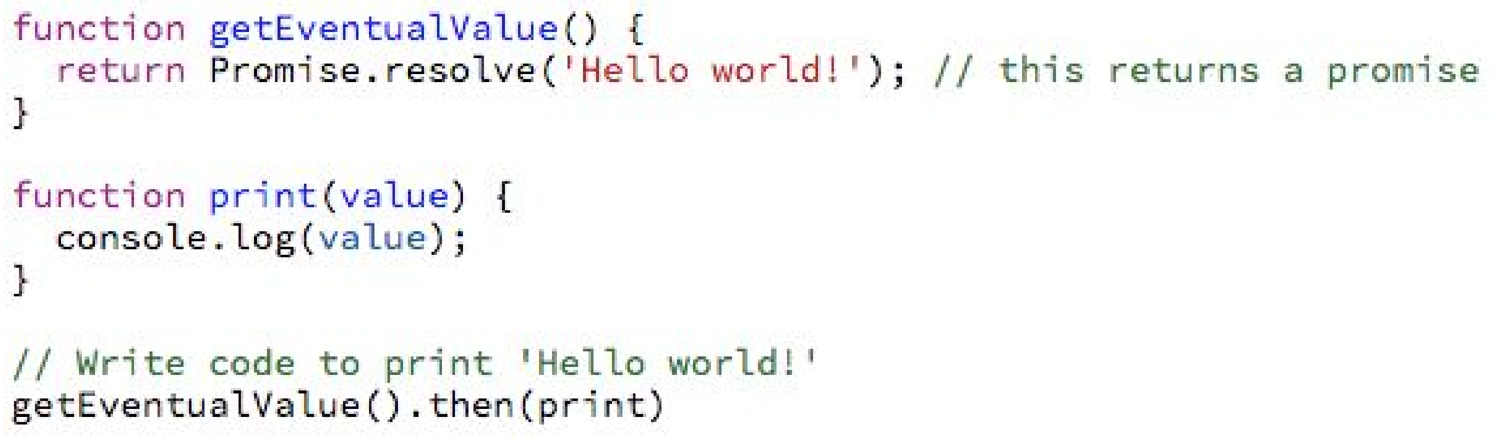
goo.gl/RX51LU



```
function getEventualValue() {
  return Promise.resolve('Hello world!'); // this returns a promise
}

function print(value) {
  console.log(value);
}

// Write code to print 'Hello world!'
```

Exercise 1

goo.gl/RX51LU



```
function getEventualValue() {
  return Promise.resolve('Hello world!'); // this returns a promise
}

function print(value) {
  console.log(value);
}

// Write code to print 'Hello world!'
getEventualValue().then(print)
```

Promise API Deep Dive

Outline

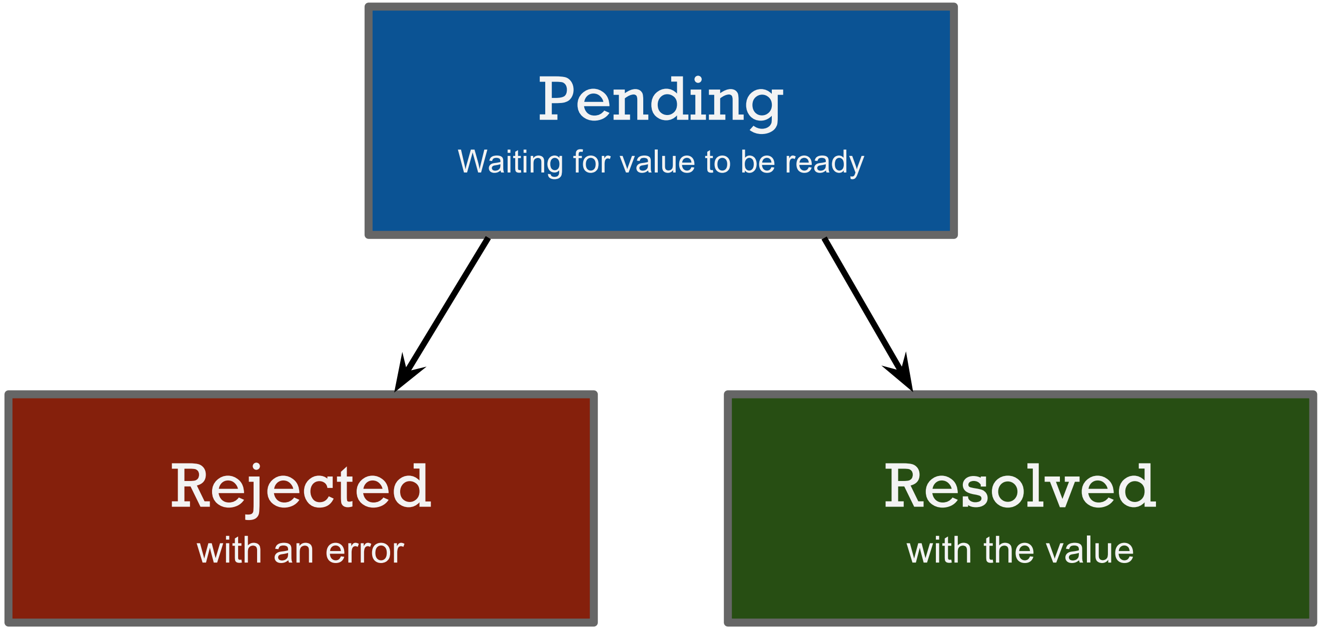
- Promise.then
- Promise.catch
- Promise.all
- Promise constructor

Promise.then()

Promise.then()

Allows you to attach handlers to Promises.

then(successHandler, errorHandler)



errorCallback is run

successCallback is run

Example

```
var promise = Promise.resolve(5)

promise.then(function (data) {
  // Success handler
  console.log(data)
}, function (error) {
  // Error handler
  console.log(error)
})
```

‘5’ is printed

Example

```
var promise = Promise.reject('failure')

promise.then(function (data) {
  // Success handler
  console.log(data)
}, function (error) {
  // Error handler
  console.log(error) ‘failure’ is printed
})
```

Promise.then returns another Promise.

```
var promise = Promise.resolve(5)

function multiplyByTwo(val) {
  return val * 2
}

function print(val) {
  console.log(val)
}

promise.then(multiplyByTwo)
```

```
var promise = Promise.resolve(5)

function multiplyByTwo(val) {
  return val * 2
}

function print(val) {
  console.log(val)
}
```

```
promise.then(multiplyByTwo)
```

This returns another promise, which is resolved when multiplyByTwo returns.

```
var promise = Promise.resolve(5)

function multiplyByTwo(val) {
  return val * 2
}

function print(val) {
  console.log(val)
}

var anotherPromise = promise.then(multiplyByTwo)
var promise = Promise.resolve(5)

function multiplyByTwo(val) {
  return val * 2
}

function print(val) {
  console.log(val)
}

var anotherPromise = promise.then(multiplyByTwo)

anotherPromise.then(print)
```

```
var promise = Promise.resolve(5)

function multiplyByTwo(val) {
  return val * 2
}

function print(val) {
  console.log(val)
}

promise.then(multiplyByTwo).then(print)
```

You can chain handlers with “then”.

```
var promise = Promise.resolve(5)

promise
  .then(multiplyByTwo)
  .then(multiplyByTwo)
  .then(multiplyByTwo)
  .then(print) // prints 40
```

```
var promise = Promise.resolve(5)

promise
  .then(multiplyByTwo)
  .then(multiplyByTwo)
  .then(multiplyByTwo)
  .then(print) // prints 40

promise
  .then(print) // prints 5
```

```
var promise = Promise.resolve(5)

function eventualMultiplybyTwo(val) {
  return Promise.resolve(val * 2)
}

function print(val) {
  console.log(val)
}

var anotherPromise = promise.then(eventualMultiplyByTwo)

anotherPromise.then(print)
```

Return a promise

What you expect...

```
var promise = Promise.resolve(5)

function eventualMultiplybyTwo(val) {
  return Promise.resolve(val * 2)
}
```

Return a promise

```
function print(val) {
  console.log(val)
}
```

val is a promise

```
var anotherPromise = promise.then(eventualMultiplyByTwo)
```

Resolved with a promise.

```
anotherPromise.then(print)
```

```
var promise = Promise.resolve(5)

function eventualMultiplybyTwo(val) {
  return Promise.resolve(val * 2)
}

function print(val) {
  console.log(val)
}

var anotherPromise = promise.then(eventualMultiplyByTwo)

anotherPromise.then(print)
```

Mirrors behavior

```
var promise = Promise.resolve(5)

function eventualMultiplybyTwo(val) {
  return Promise.resolve(val * 2)
}

function print(val) {
  console.log(val)   val = 10
}

var anotherPromise = promise.then(eventualMultiplyByTwo)

anotherPromise.then(print)
```

You can return a Promise in a successHandler, and the next “then” will receive the resolved value.

(MAGIC!)

Useful for multi-step fetches

```
function getFavoriteBook() {
  return getWithPromise('/user/favorite-book')
}

function getBookInfo(bookId) {
  return getWithPromise('/book/' + bookId + '/info')
}

getFavoriteBook().then(getBookInfo).then(print)
```

Summary of then

- You can chain handlers with “then”.
- You can return Promises from handlers instead of values.

Promise.catch()

Promise.catch()

Allows you to handle rejected promises.

catch(errorHandler) is the same as then(undefined, errorHandler)

If a Promise is rejected, all then’s are skipped until the next catch.

```javascript
var promise = Promise.resolve(5)

promise.then(function(val) {
  throw new Error('Oh no!')
}).then(function(val) {
  console.log('This will not be printed')
}).catch(function(error) {
  console.log(error.message) // prints 'Oh no!'
})
```

After a catch, the promise chain recovers.

```
var promise = Promise.resolve(5)

promise.then(function(val) {
  throw new Error('Oh no!')
})
.catch(function(error) {
  console.log(error.message) // prints 'Oh no!'
  return 'Recovered'
})
.then(function(val) {
  console.log(val) // prints 'Recovered'
})
.catch(function () {
  console.log('Not called because chain has recovered')
})
```

```
var promise = Promise.resolve(5)

promise.then(function(val) {
  throw new Error('Oh no!')
})
.catch(function(error) {
  console.log(error.message) // prints 'Oh no!'
  throw error
})
.then(function() {
  console.log('Not called because error was re-thrown')
})
```

Summary of catch

- After a catch, the promise chain recovers.
- To prevent a recovery, throw another error.

Exercise 2

goo.gl/v3zZeZ

```
var promiseA = Promise.resolve(5)
var promiseB = promiseA.then(function (val) {
  return val * 2;
})

promiseB.then(function (val) {
  console.log('then1: ' + val)
  return val
})
.then(function (val) {
  throw {
    'message': 'Oh no!',
    'val': val,
  }
})
.then(function (val) {
  console.log('then2: ' + val)
  return val
})
.catch(function (error) {
  console.log('catch1: ' + error.message)
  return promiseA.then(function (val) {
    return val * error.val
  })
})
.then(function (val) {
  console.log('then3: ' + val)
  return val
})
.catch(function (error) {
  console.log('catch2: ' + error.message)
})

// What's the output?
```

Exercise 2

goo.gl/v3zZeZ

```
var promiseA = Promise.resolve(5)
var promiseB = promiseA.then(function (val) {
  return val * 2;
})

promiseB.then(function (val) {
  console.log('then1: ' + val)
  return val
})
.then(function (val) {
  throw {
    'message': 'Oh no!',
    'val': val,
  }
})
.then(function (val) {
  console.log('then2: ' + val)
  return val
})
.catch(function (error) {
  console.log('catch1: ' + error.message)
  return promiseA.then(function (val) {
    return val * error.val
  })
})
.then(function (val) {
  console.log('then3: ' + val)
  return val
})
.catch(function (error) {
  console.log('catch2: ' + error.message)
})

// What's the output?
```

```
"then1: 10"
"catch1: Oh no!"
"then3: 50"
```

Promise.all()

Promise.all()

Allows you to combine multiple promises.

```
var promiseA = Promise.resolve(5)
var promiseB = Promise.resolve(2)

Promise.all([promiseA, promiseB]).then(function (val) {
  console.log(val) // prints [5, 2]
})
```

Many more Promise functions in libraries like Bluebird or Q.

Promise constructor

Promise constructor

For creating your own custom promises.

Constructor takes executor function:

```
function (resolve, reject) { … }
```

```
var inFiveSeconds = new Promise(function (resolve, reject) {
  setTimeout(function() {
    resolve(5)
  }, 5000)
})
```

```
var inFiveSeconds = new Promise(function (resolve, reject) {
  setTimeout(function() {
    resolve(5)
  }, 5000)
})
```

```
var inFiveSeconds = new Promise(function (resolve, reject) {
  setTimeout(function() {
    resolve(5)
  }, 5000)
})
```

Resolves promise once eventual value is ready.

Implementing getWithPromise

```
$.get('/api/get_data', successCallback, errorCallback)

var promise = getWithPromise('/api/get_data')

// Implement getWithPromise by wrapping $.get
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {









}
```

```
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {
  return new Promise(function (resolve, reject) {








  })
}
```

```
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {
  return new Promise(function (resolve, reject) {
    var successCallback = function(val) {

    }





  })
}
```

```
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {
  return new Promise(function (resolve, reject) {
    var successCallback = function(val) {
      resolve(val)
    }





  })
}
```

```
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {
  return new Promise(function (resolve, reject) {
    var successCallback = function(val) {
      resolve(val)
    }
    var errorCallback = function(error) {

    }

  })
}
```

```
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {
  return new Promise(function (resolve, reject) {
    var successCallback = function(val) {
      resolve(val)
    }
    var errorCallback = function(error) {
      reject(error)
    }

  })
}
```

```javascript
// $.get('/api/get_data', successCallback, errorCallback)

function getWithPromise(url) {
  return new Promise(function (resolve, reject) {
    var successCallback = function(val) {
      resolve(val)
    }
    var errorCallback = function(error) {
      reject(error)
    }
    $.get(url, successCallback, errorCallback)
  })
}
```

Summary

- Promise.then
- Promise.catch
- Promise.all
- Promise constructor

Why are Promises cool?

Promises are more flexible than CPS.

When the eventual value is resolved…

```
$.get('/api/get_data', successCallback)
```

```
var promise = getWithPromise('/api/get_data')
```

When the eventual value is resolved...

```
$.get('/api/get_data', successCallback)
```

```
var promise = getWithPromise('/api/get_data')
```

- I can handle it with successCallback.

When the eventual value is resolved...

```
$.get('/api/get_data', successCallback)
```

- I can handle it with successCallback.

```
var promise = getWithPromise('/api/get_data')
```

- I can apply multiple handlers to it to produce derived values.
- I can save it for later.
- I can combine multiple promises.
- ...And much more.

When the eventual value is resolved...

```
$.get('/api/get_data', successCallback)
```

- I can handle it with successCallback.

```
var promise = getWithPromise('/api/get_data')
```

- I can apply multiple handlers to it to produce derived values.
- I can save it for later.
- I can combine multiple promises.
- ...And much more.

With sufficiently complex callbacks, you can technically do everything you can with Promises.

As situations grow complex, Promises produce cleaner, shorter code than CPS.

```
var promiseA = Promise.resolve(5)
var promiseB = Promise.resolve(2)

Promise.all([promiseA, promiseB]).then(function (val) {
  console.log(val) // prints [5, 2]
})
```

```
var getA = function (successCallback) {...}
var getB = function (successCallback) {...}

var a, b

getA(function(val) {
  a = val
  if (!!a && !!b) {
    console.log([a, b])
  }
})

getB(function(val) {
  b = val
  if (!!a && !!b) {
    console.log([a, b])
  }
})
```

Use Promises for your GET and POST requests and save yourself future pain.

Outline

- What are Promises?
- Promise API deep dive
- Why are Promises cool?
- Further topics to explore

Further Topics to Explore

Check out window.fetch for Promise-based AJAX requests.

Promises are supported natively, but check out Bluebird or Q for a fuller API.

Further topics to explore

- Async-await functions
- ES6/Babel
- Observables (RxJS)

Thanks!

Intro to Javascript Promises

Mark Zhang

KENSHO

6.148 MIT Web Programming Competition 2017