Promises

How a Promise works

A Promise represents a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a promise of having a value at some point in the future. [1]

[1] Mozilla Developer Network

In other words…

- Allows you to create functions that return a value that does not exist yet
- Gives you a way to create easy to read asynchronous code
- Wraps asynchronously executing code into chains of function calls to make things more legible
- Simplifies error handling and error bubbling

Promise States

- Promises can exist in one of three states:
  - `pending`: The initial state of a Promise upon creation.
  - `fulfilled`: The pending operation has completed successfully.
  - `rejected`: The pending operation failed.

Other Terminology You Might See

- resolved: a promise that has fulfilled a value
- settled: a promise that has either been fulfilled or rejected

Creating a Promise

- New promises take a function that has two function arguments - resolve and reject
- Promise functions are intended to do some work, *eventually* calling resolve or reject

```
var promise = new Promise(function(resolve, reject) {
  // perform an action
  // this could be any synchronous or asynchronous operation

  // if there was an error
  reject(error);

  // if everything went well
  resolve(result);
});
```

Using a Promise: `promiseInstance.then()`

- All promise instance get a `.then()` method that allow you to handle promise resolution
- The `.then()` callback receives the result given by what was passed to the promise's `resolve`
- Once a promise is fulfilled (when `resolve` is called), the fulfillment value is passed to the `.then()` handler

Example: Using `promiseInstance.then()`

```
// create a promise
var promise = new Promise(function (resolve, reject) {
  // after a waiting a second,
  setTimeout(function() {
    // resolve the promise with a message
    resolve('hello world!');
  }, 1000);
});

promise.then(function(result){
  // prints "hello world!" after 1s
  console.log(result);
});
```

Using a Promise: `promiseInstance.catch()`

- All promise instances also get a `.catch()` method that allow you to handle promise rejection
- The `.catch()` callback receives the result given by what was passed to the promise's `reject`
- Typically, `.catch()` is called with an `Error` (or a type that inherits from `Error`)

Example: Using `promiseInstance.catch()`

```
// create a promise
var promise = new Promise(function (resolve, reject) {
  // after a waiting a second,
  setTimeout(function() {
  // reject the promise with an Error
    reject(new Error('uh oh!'));
  }, 1000);
});

promise.catch(function(err){
  // prints the error "uh oh!" with the stack after 1s
  console.log(err);
});
```

Using a Promise: `Promise.resolve()`

- Instead of creating a Promise, you can use the static method `.resolve()` to fulfill values
- Makes a Promise that is already resolved to the value you pass

Example: Using `Promise.resolve()`

```
// create a promise using `.resolve()`
var promise = Promise.resolve('hello world!');

promise.then(function(result){
  // prints "hello world!"
  console.log(result);
});
```

Using a Promise: `Promise.reject()`

- Again, instead of creating a Promise, you can use the static method `.reject()`to create a rejected Promise
- Like `.resolve()`, this creates a promise that is already in a rejected state, passing the error

Example: Using `Promise.reject()`

```
// create a promise using `.reject()`
var promise = Promise.reject(new Error('uh oh'));

promise.catch(function(err){
  // prints "hello world!"
  console.log(err);
});
```

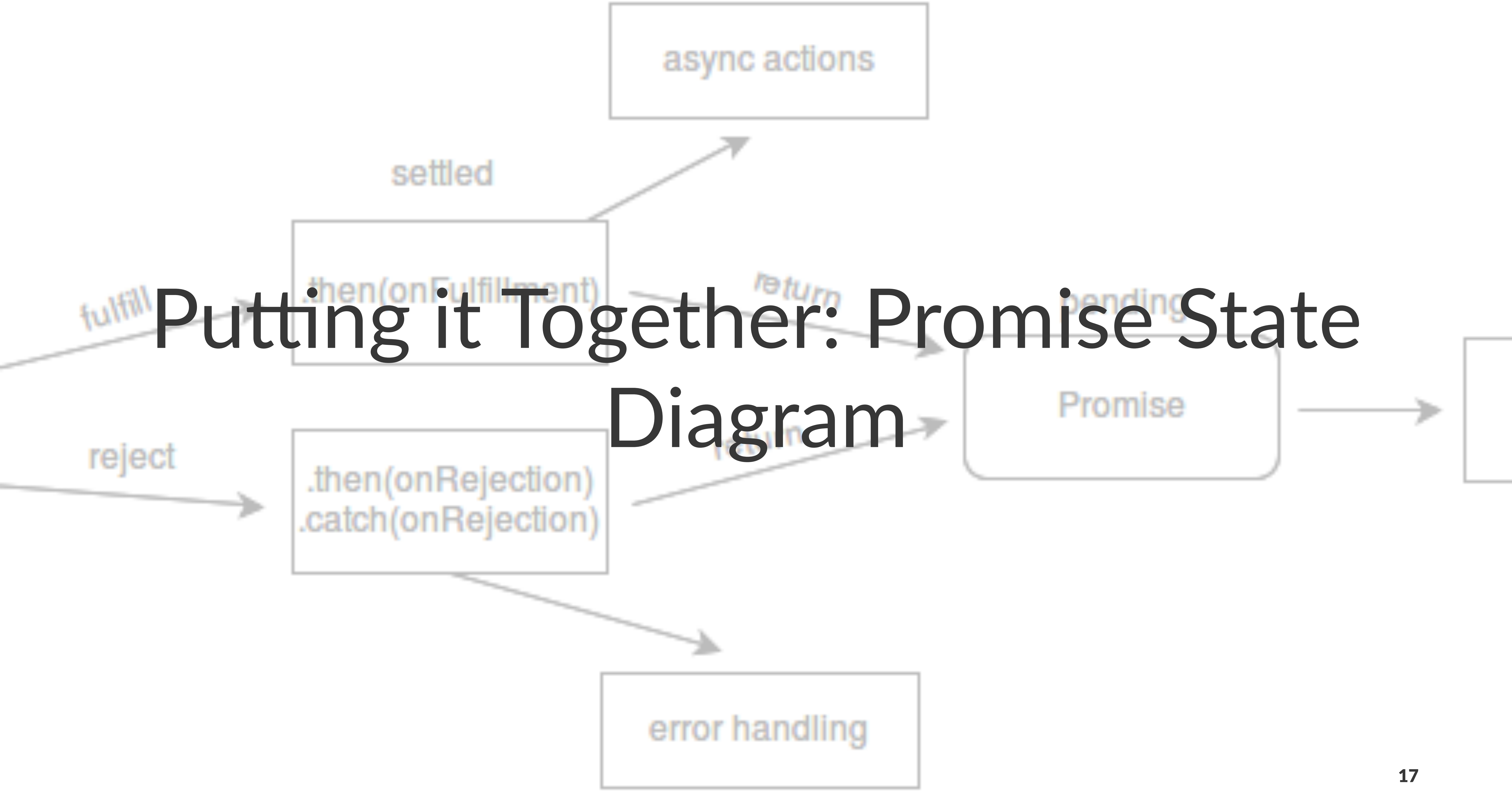
Function Composition/Chaining

- `.then()` and `.catch()` both return Promises
- This means that the return values can call `.then()` and `.catch()` as well, chaining the methods together in an operation known as composition
- Chaining allows Promises to allow for more complex behavior
- Note that values must be returned from the `.then()` handler
- Returned values can be promises or any other object, function, or primitive

Example: Chained `then` call

```
var promise = Promise.resolve('hello')
  .then(function(str) {
    return new Promise(function(resolve, reject) {
      setTimeout(function() {
        resolve(`${str} there`);
      }, 1000);
    });
  })
  .then(function(str) {
    return `${str} world!`;
  })
  .then(function(str) {
    //prints "hello there world!"
    console.log(str);
    //the `promise` variable above will eventually
    //be a fulfilled promise with `str` as its value
    return Promise.resolve(str);
  });
```

Putting it Together: Promise State Diagram



What if All I Have is Callbacks?

- Not all libraries or modules have a Promise-based API
- Sometimes, it is helpful to wrap callbacks with promises

`Promise.all(Array|Iterable)`

- Waits for all promises to finish
- Returns a promise, so it is chainable, just like `.then()` and `.catch()`
- Results will be an array in the same order as the array given to `.all()`

Example: `Promise.all()`

```
//create an array.  We're going to use this array like a queue.
var promise1 = Promise.resolve('one');
var promise2 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve('two, with some time');
  }, 2000);
});
var promise3 = new Promise(function(resolve, reject) {
  resolve('three');
});
//`Promise.all` waits for all of the promises to complete
Promise.all([promise1, promise2, promise3])
  .then(function(results) {
    //prints ['one', 'two, with some time', 'three']
    //this will happen after 2 seconds as `promise2`
    //takes time to resolve
    console.log(results);
  });
```