

Mechanics of Promises

Understanding Javascript Promise Generation & Behavior



Async: continuation-passing

```
// Mongoose & Express
models.Page.findOne({_id: id}).exec(function(err, page){
  if (err) res.status(500).end();
  res.json(page);
});
```



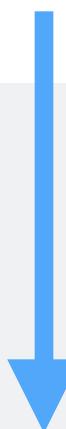
Async: promise

```
// Mongoose & Express
models.Page.findOne({_id: id}).exec().then(
  function (page) { res.json(page); },
  function (err) { res.status(500).end(); }
);
```

Why?



Async: promise



```
// Mongoose & Express
models.Page.findOne({_id: id}).exec().then(
  function (page) { res.json(page); },
  function (err) { res.status(500).end(); }
);
```



Break free from the async call!

```
var promise = models.Page.findOne({_id: id}).exec();
// promise is portable – can move it around
promise.then(
  function (page) { res.json(page); },
  function (err) { res.status(500).end(); }
);
```



Export to other modules...

```
var presidentPromise = models.User.findOne({ role: 'president' }).exec();
module.exports = presidentPromise;
```



...collect in arrays and pass into functions...

```
var dayPromises = [];
// make 7 parallel (simultaneous) day requests
for (var i = 0; i < 7; i++) {
  var promiseForDayI = models.Day.findOne({dayNum: i}).exec();
  dayPromises.push( promiseForDayI );
}
// act only when they have all resolved
Q.all( dayPromises ).then(function(days){
  res.render('calendar', {days: days});
});
```



...and much more

```
promiseForUser
  .then( function (user) {
    return promiseForMessages = asyncGet(user.messageIDs);
  })
  .then( function (messages) {
    return promiseForComments = asyncGet(messages[0].commentIDs);
  })
  .then( function (comments) {
    UI.display( comments[0] );
  })
  .catch( function (err) {
    console.log('Fetch error: ', err);
  })
  .done();
```

So, what is a promise?

*“A promise represents the eventual result
of an asynchronous operation.”*

— THE PROMISES/A+ SPEC



magic!
(no)

Promises are Objects

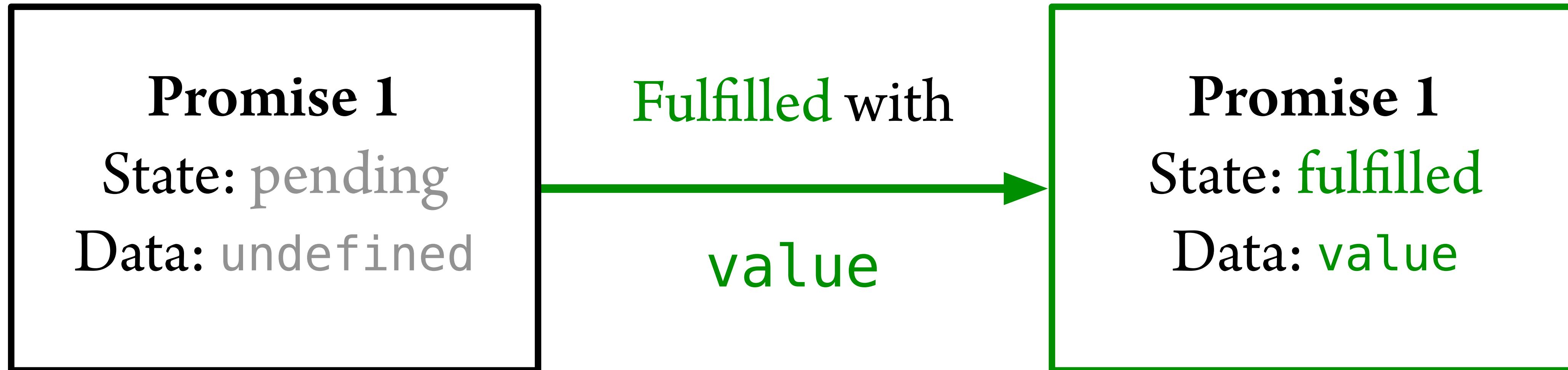
state (pending, fulfilled, or rejected)

information (value or a reason)

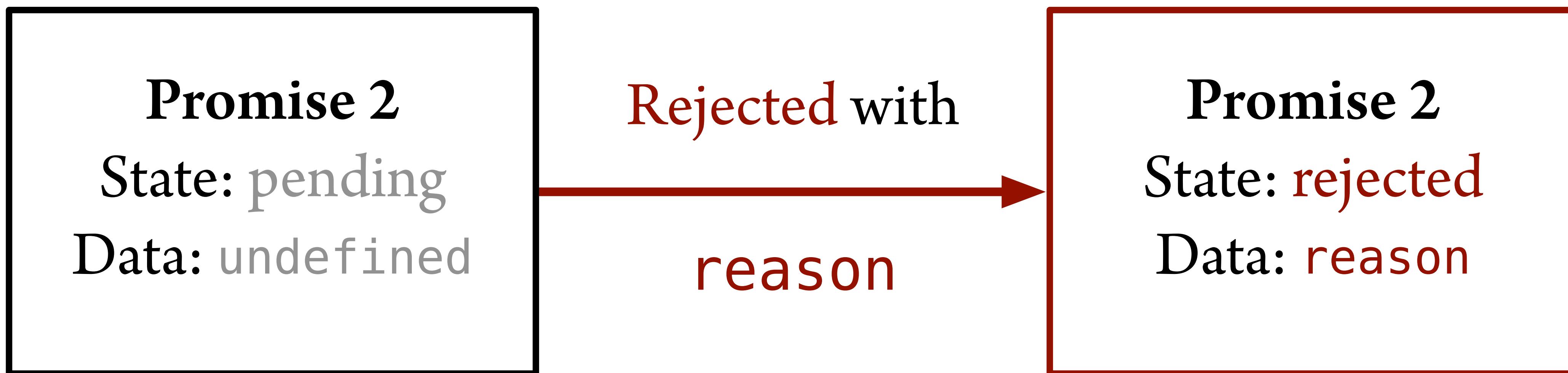
.then()

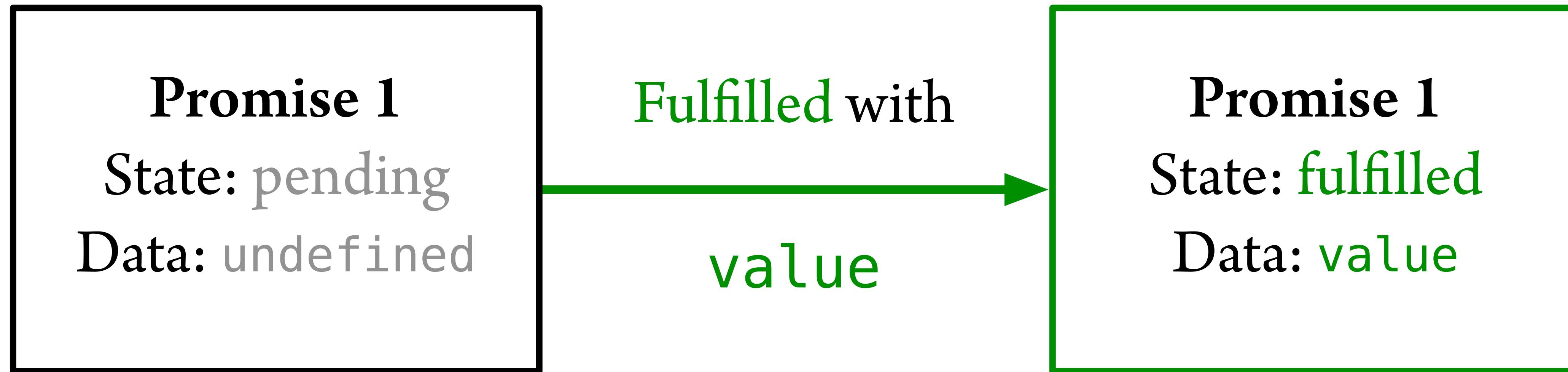
(via closure)

(property)

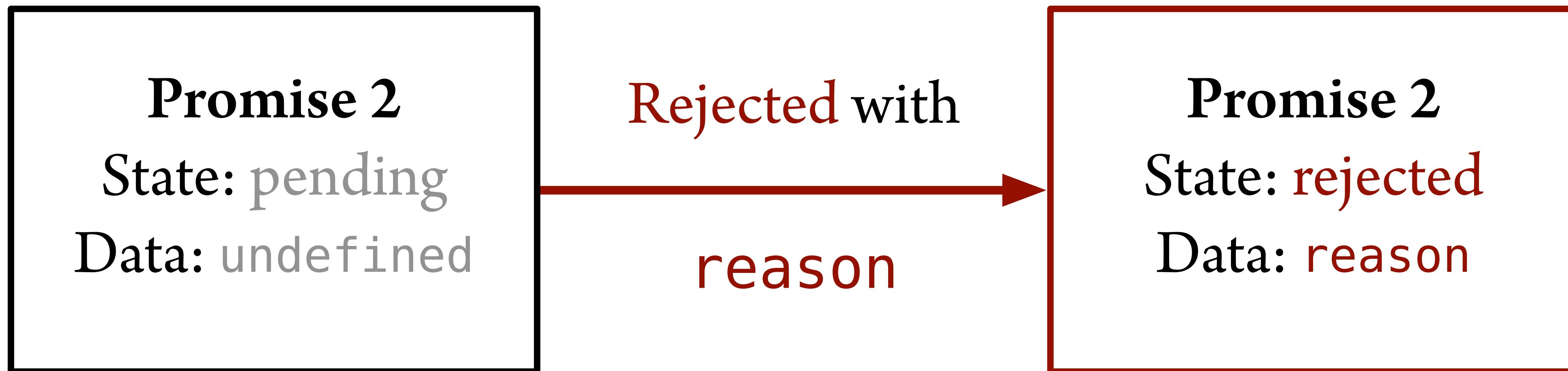


promises only change state while pending





`aPromise.then(successHandler, failureHandler)`



Timing-ambivalent

- 1. Add *handler*
2. **promise settles**
3. handler is called once
- 1. **Promise settles**
2. *add handler*
3. handler is called once
- Can attach handlers at multiple times (different modules even)

```
userPromise.then(welcomeUser);
```

```
userPromise.then(showCart);
```

What does this solve?

“The point of promises is to give us back functional composition and error bubbling in the async world.”

– DOMENIC DENICOLA, “YOU'RE MISSING THE POINT OF PROMISES”



Callback Hell

deep, confusing nesting & forced, repetitive error handling

```
// Basic async callback pattern.  
// asyncFetchUser asks a server for some data.  
// Internally, it gets a response: { name: 'Kim' }.  
// That response is then passed to the receiving callback.
```

```
asyncFetchUser( 123, function received( response ) {  
  console.log( response.name ); // output: Kim  
});
```

```
// Callback Hell

var userID = 'a72jd3720h';
getUserData( userID, function got( userData ) {
  getMessage( userData.messageIDs[0], function got( message ) {
    getComments( message, function got( comments ) {
      console.log( comments[0] );
    });
  });
});
```

```
// Callback Hell... with error handling, for extra hellishness

var userID = 'a72jd3720h';
getUserData( userID, function got( err, userData ) {
  if (err) console.log('user fetch err: ', err);
  else getMessage( userData.messageIDs[0], function got( err, message ) {
    if (err) console.log('message fetch err: ', err);
    else getComments( message, function got( err, comments ) {
      if (err) console.log('comment fetch err: ', err);
      else console.log( comments[0] );
    });
  });
});
```

```
promiseForUser
  .then(function (user) {
    return asyncGet(user.messageIDs);
  })
  .then(function (messages) {
    return asyncGet(messages[0].commentIDs);
  })
  .then(function (comments) {
    UI.display( comments[0] );
  })
  .catch(function (err) {
    console.log('Fetch error: ', err);
  })
  .done();
```



Black Holes

you can literally not return from a typical async callback

```
// This will not work!
```

```
var person = asyncGetGroup( 123, function got( group ) {  
  return group.users[0];  
});
```

```
// This also will not work

var person;

asyncGetGroup( 123, function got( group ) {
  person = group.users[0];
});

// somewhere else
var headline = person.name; // might be undefined!
```

```
// Fantasy solution

var containerA = new Container();
asyncGetData( function got( data ) {
    containerA.save( data ); // once async completes
});

// ...somewhere else...
containerA.whenSaved( function use( data ) {
    console.log( data ); // once containerA.save() happens
});
```



That's a promise

Complex landscape... but standardized

- Multiple proposed standards • CommonJS
- One leading standard • Promises/A+
- Upcoming native ES6 promises (already working in some browsers)
- Two different approaches for generating new promises
 - CommonJS-style *deferrals* (one extra entity)
 - ES6-style *constructors* (simplified)
- **jQuery gurus beware! `$.Deferred` differs from current standards and is considered flawed. See Kris Kowal's guide.**

So where do real promises come from?

- Existing libraries may return promises
 - Angular \$http
 - Mongoose .exec()
- Wrap async calls in promise-makers
 - Q / \$q deferral
 - ES6 / Bluebird constructor
- Promise libraries can wrap for us, e.g. in Node
 - Q.denodeify(fs.readFile)
 - Bluebird.promisifyAll(fs)



Promisification in Node.js

```
fs.readFile('foo.txt', 'utf-8', function (err, text) {  
  // use the text  
});
```

```
var readFile = Q.denodeify( fs.readFile );  
readFile('foo.txt', 'utf-8').then(function (text) {  
  // use the text  
});
```

```
Bluebird.promisifyAll( fs );  
fs.readFileAsync('file.j', 'utf8').then(function (text) {  
  // use the text  
});
```



Making Promises: Deferral? Constructor?



Constructor-style promise generation

```
var myPromise = new Promise( function (resolve, reject) {  
    someAsyncCall( function (err, data) {  
        if (err) reject( err );  
        else resolve( data );  
    });  
});  
  
// elsewhere  
myPromise.then( someSuccessHandler, someErrorHandler );
```

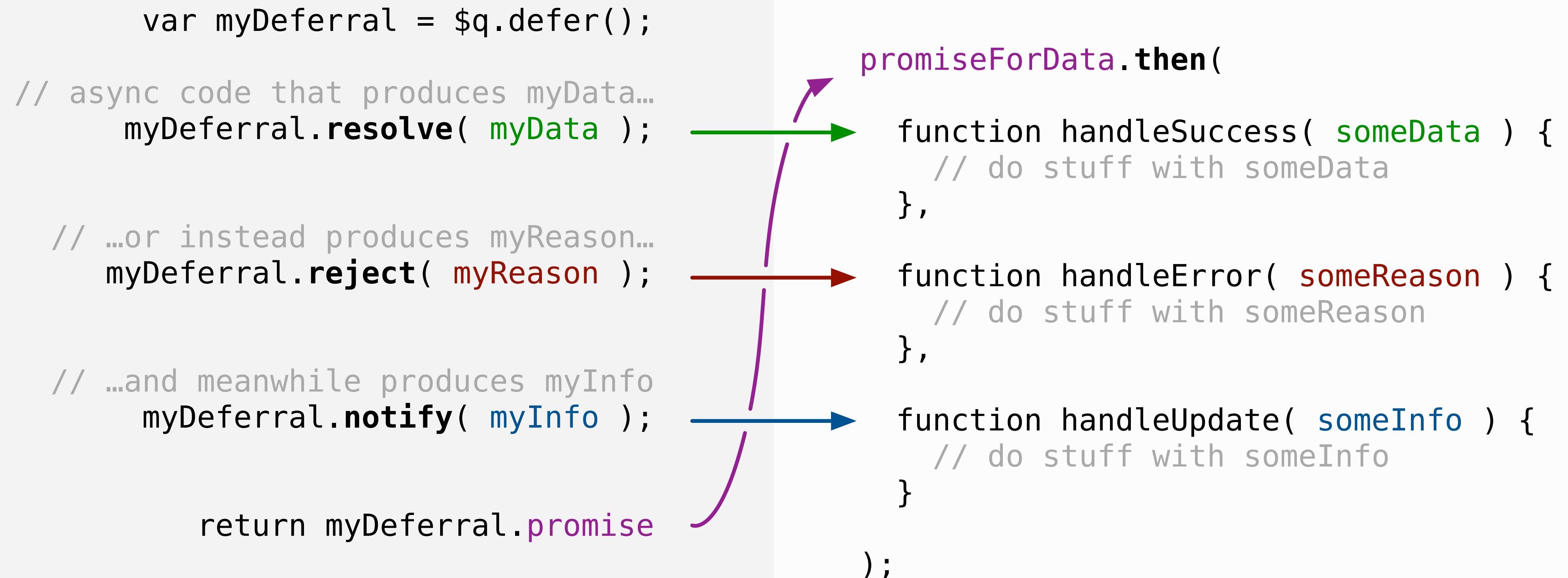


Deferral-style promise generation

```
var myDeferral = $q.defer();
someAsyncCall( function (err, data) {
  if (err) myDeferral.reject( err );
  else myDeferral.resolve( data );
});
var myPromise = myDeferral.promise;

// elsewhere
myPromise.then( someSuccessHandler, someErrorHandler );
```

A deferral controls its promise's state. Promises attach callbacks to an eventual value.



The async code can exist in a service...

... and return a promise for myData to a controller!

(Break)



the magic: .then returns a new promise

```
promiseB =  
promiseA.then( [successHandler], [errorHandler] );
```



This is why we can chain .then

```
promiseForThing
  .then( doStuff )
  → .then( doOtherStuff )
  → .then( doMoreStuff )
  → .catch( handleErr );
```

.**catch**(handleErr) is equivalent to .then(null, handleErr)



And why we can return from a handler

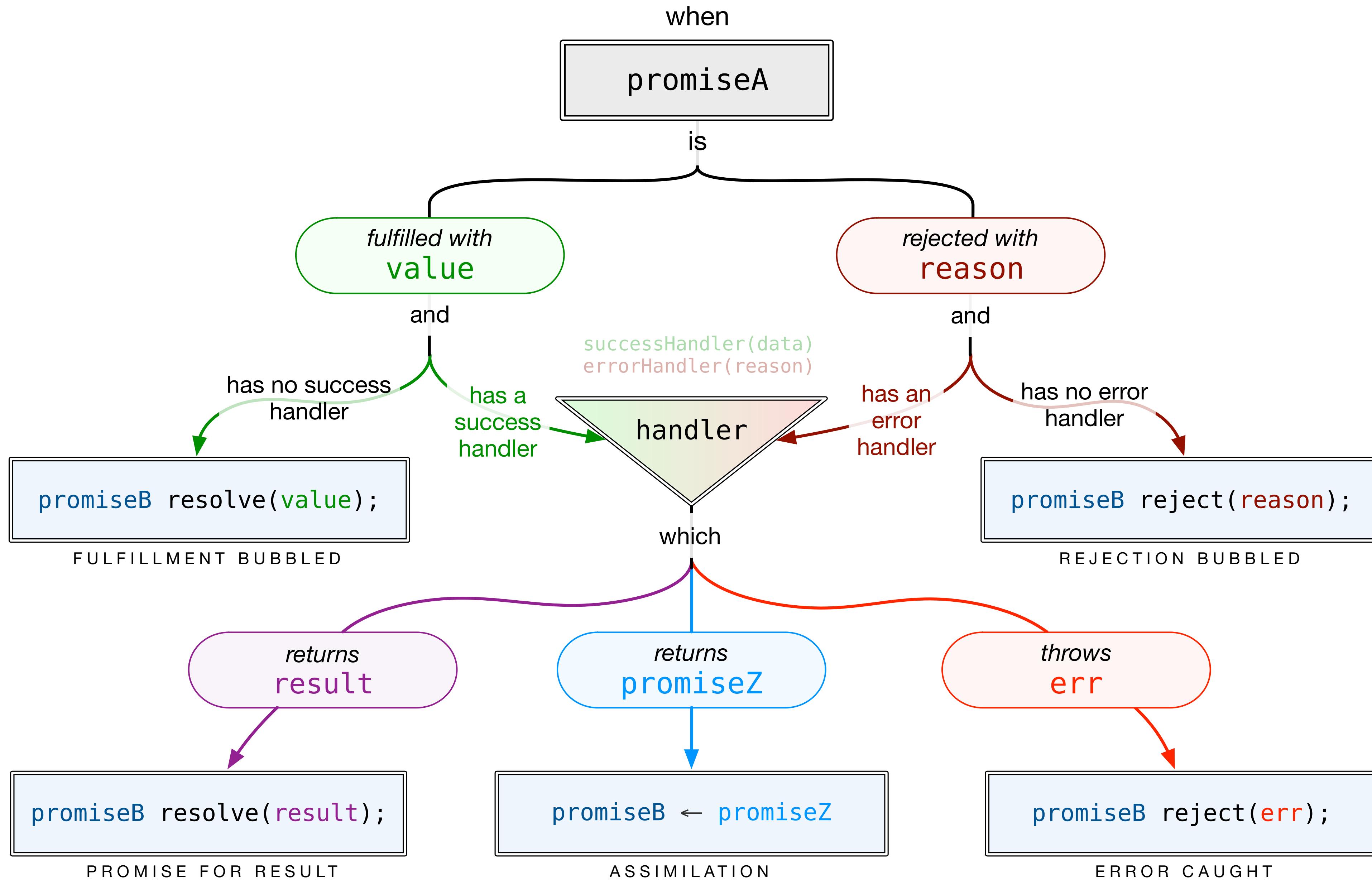
```
var promiseForThingB = promiseForThingA
  .then( function thingSuccess (thingA) {
    // run some code
    return thingB;
})
```

What is promiseB a promise for?



Brace yourselves...

```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```

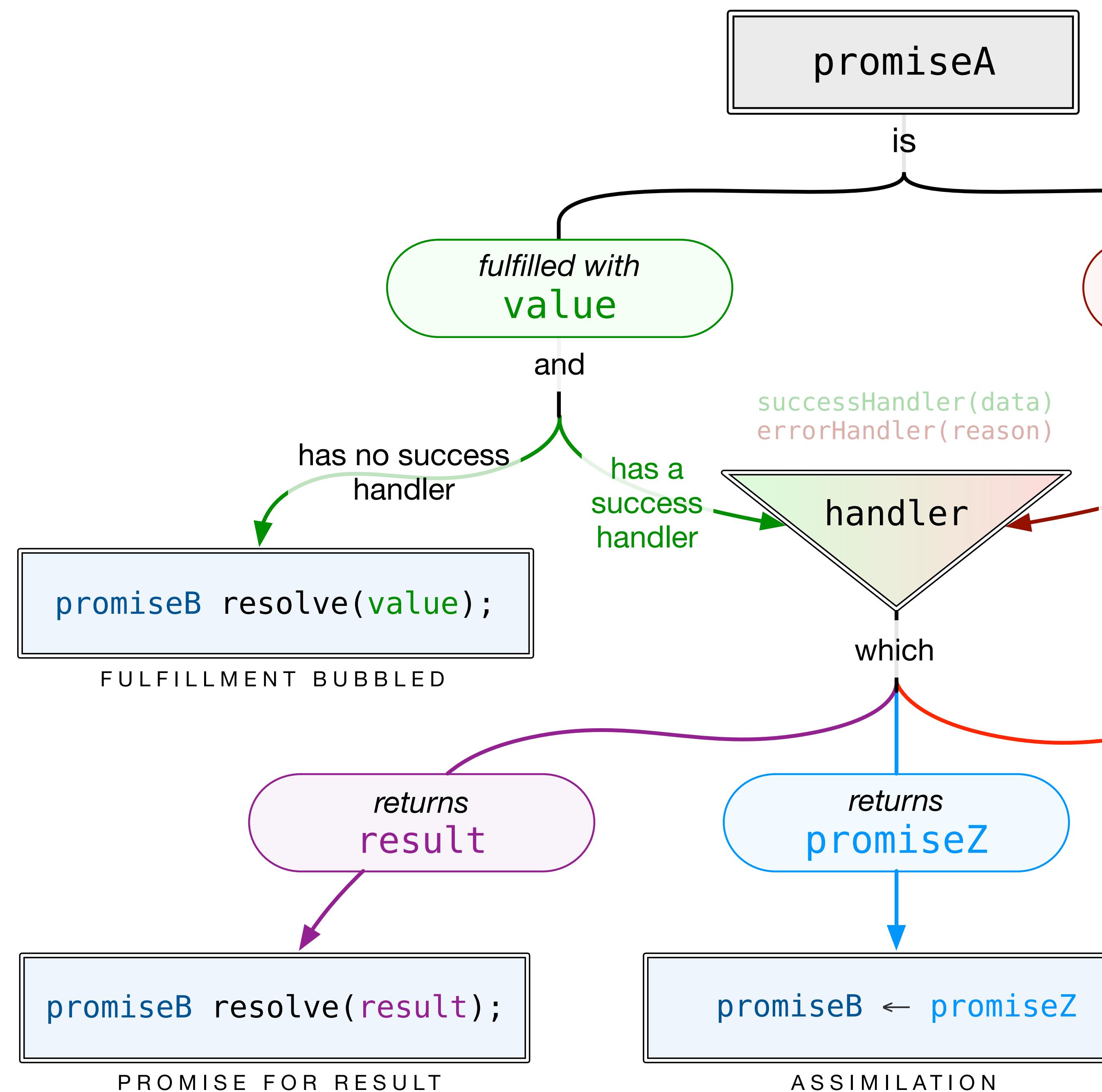


```
// promise0 fulfills with 'Hello.'
```

```
promise0
  .then() // -> p1
  .then() // -> p2
  .then() // -> p3
  .then() // -> p4
  .then() // -> p5
  .then(console.log);
```

Fulfillment bubbled down to first available success handler:

Console log reads “Hello.”

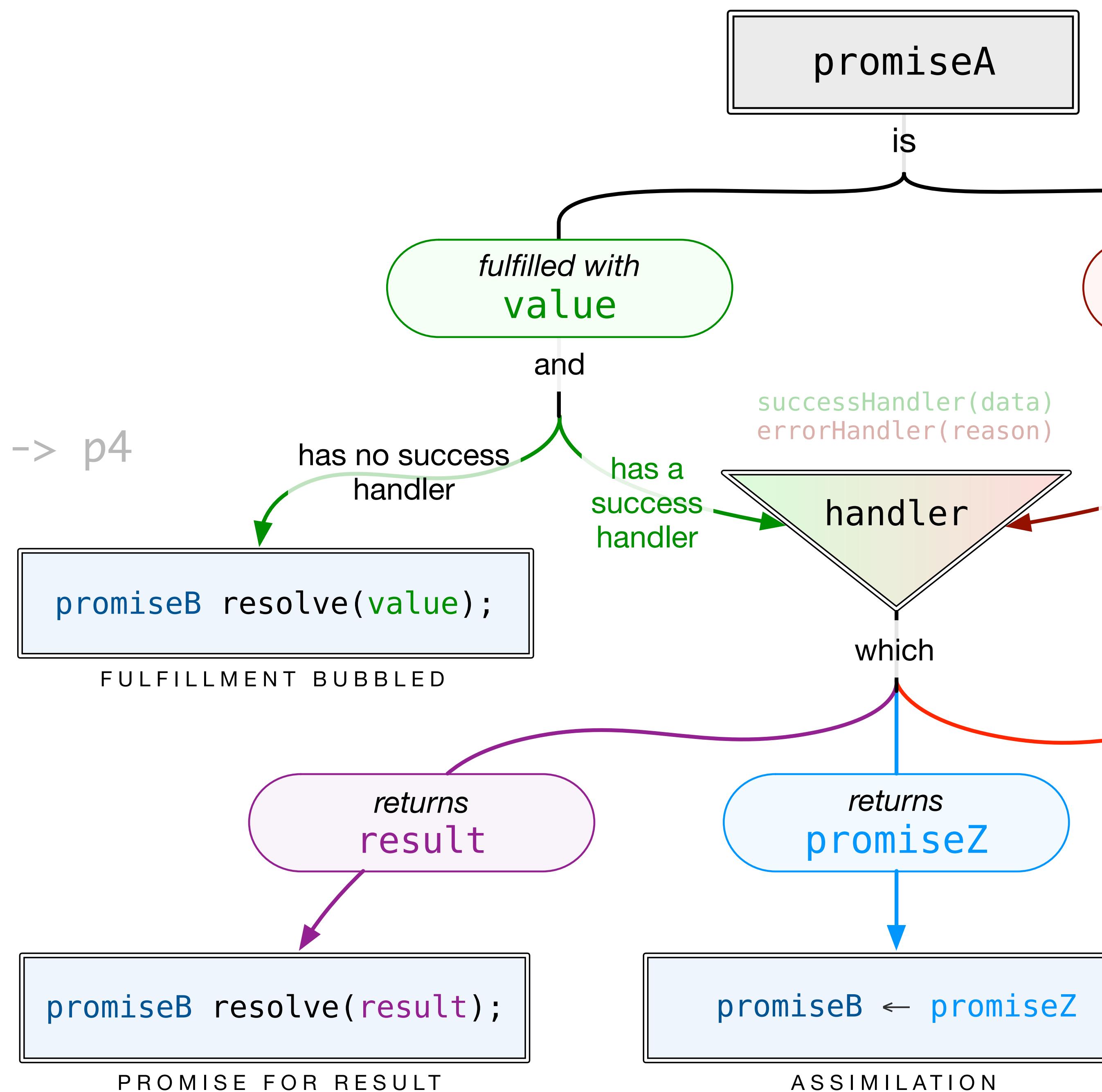


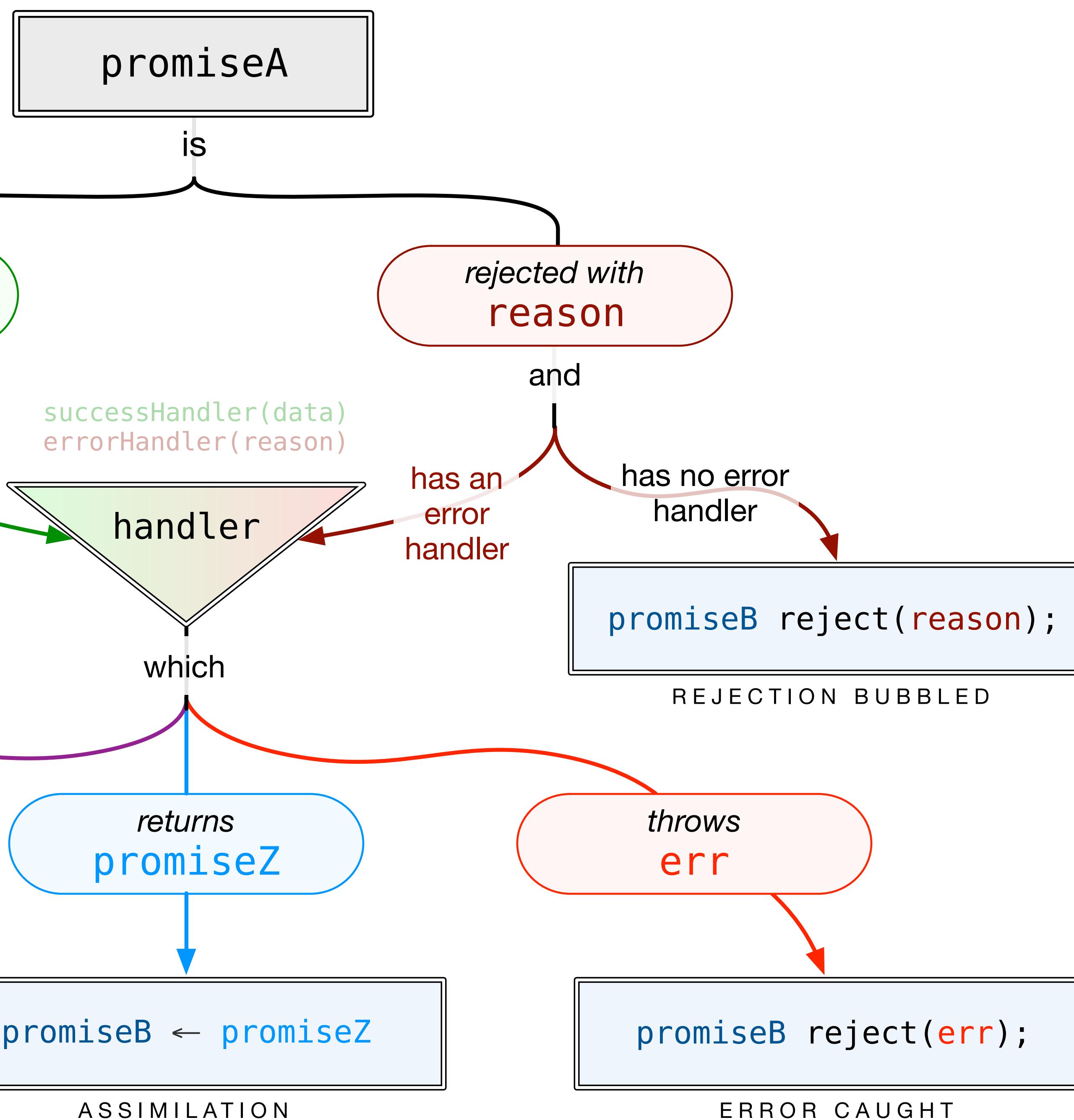
```
// promise0 fulfills with 'Hello.'
```

```
promise0
  .then(null, warnUser) // -> p1
  .then() // -> p2
  .then() // -> p3
  .then(null, null, setLoadBar) // -> p4
  .then() // -> p5
  .then(console.log);
```

Same thing! Each outgoing promise is resolved with "Hello," and each .then will pass it along unless it has a success handler.

Console log reads “Hello.”





```

function logYell (input) {
  console.log(input+'!');
}

```

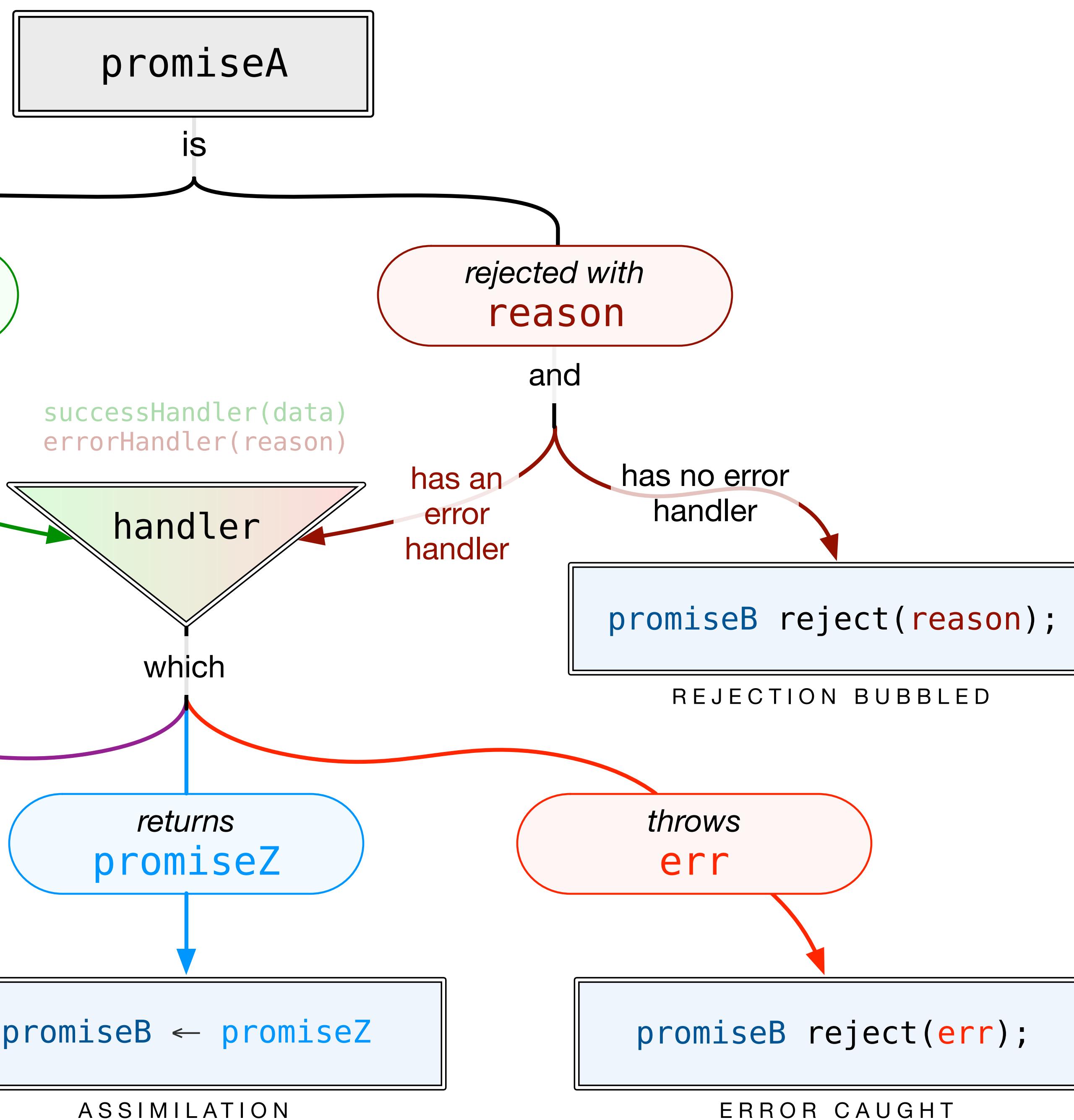
// promise0 rejected with ‘Sorry’

promise0

- `.then()` // -> p1
- `.then()` // -> p2 and so on
- `.then()`
- `.then(null, console.log);`

Rejection bubbles down to the first available error handler.

Console log is “Sorry”.



```
function logYell (input) {
  console.log(input + ' ! ');
}
```

// promise0 rejected with ‘Sorry’

promise0

```
.then(console.log) // -> p1
.then() // -> p2 and so on
.then(null, null, console.log)
.then(null, logYell);
```

Again, rejection bubbles down to the first available **error** handler.

Console log is “Sorry!”



Review: Success & Error Bubbling

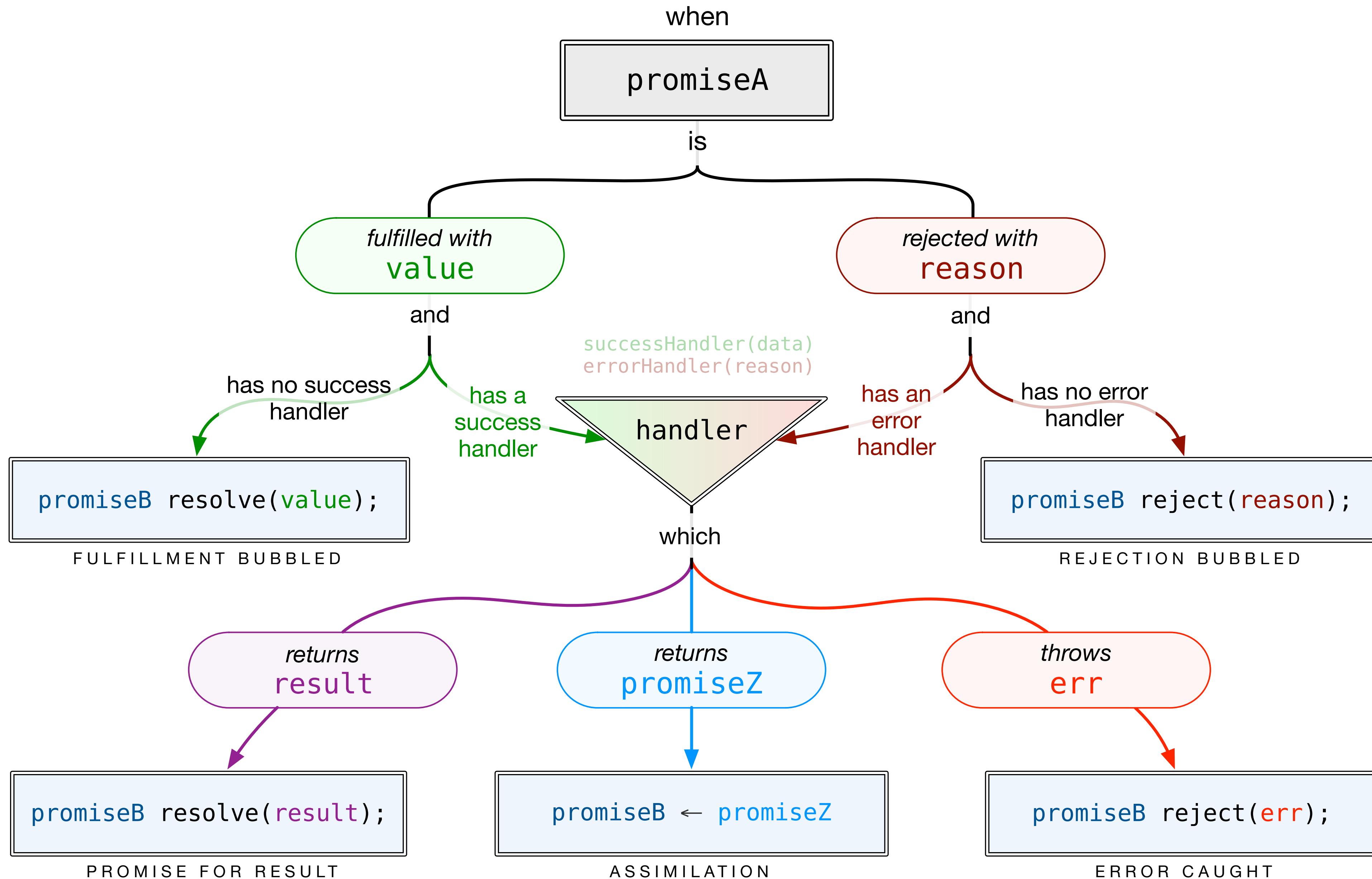
```
// promiseA is fulfilled with 'hello'  
promiseA  
  .then( null, myFunc1, myFunc2 )  
  .then()  
  .then( console.log );
```

*// result: console shows 'hello'
// fulfill bubbled to success handler*

```
// promiseA is rejected with 'bad request'  
promiseA  
  .then( myFunc1, null, myFunc2 )  
  .then()  
  .then( null, console.log );
```

*// result: console shows 'bad request'
// rejection bubbled to error handler*

```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



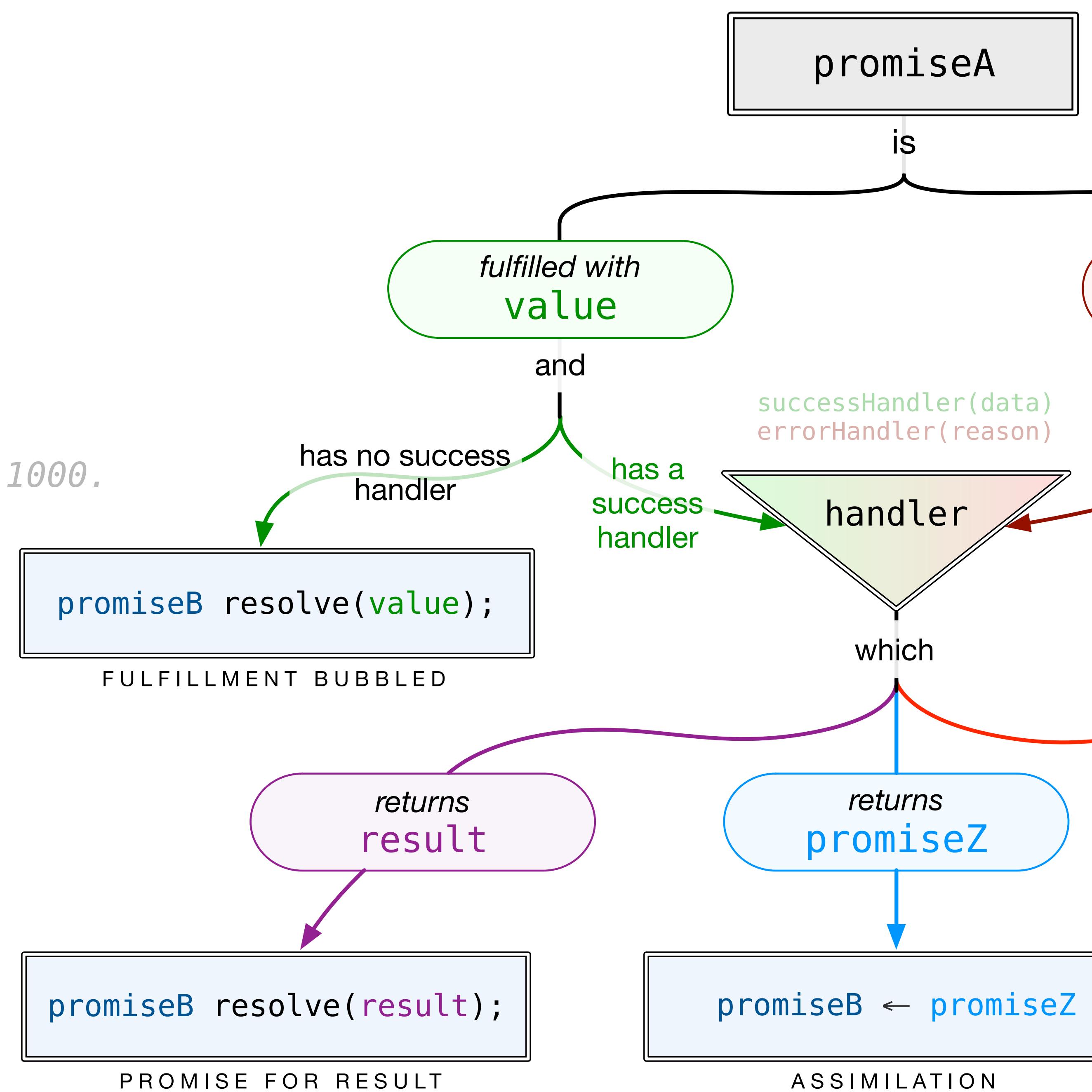
```
// promiseForVal1 is fulfilled with 999
```

```
promiseForVal2 = promiseForVal1
  .then( function success (val1) {
    val2 = val1++;
    return val2;
});
```

// result: promiseForVal2 is fulfilled with 1000.

// same idea, shown in a direct chain:

```
promiseForVal1
  .then( function success (val1) {
    // do some code to make val2
    return val2;
})
  .then( function success (val2) {
    console.log( val2 );
});
```

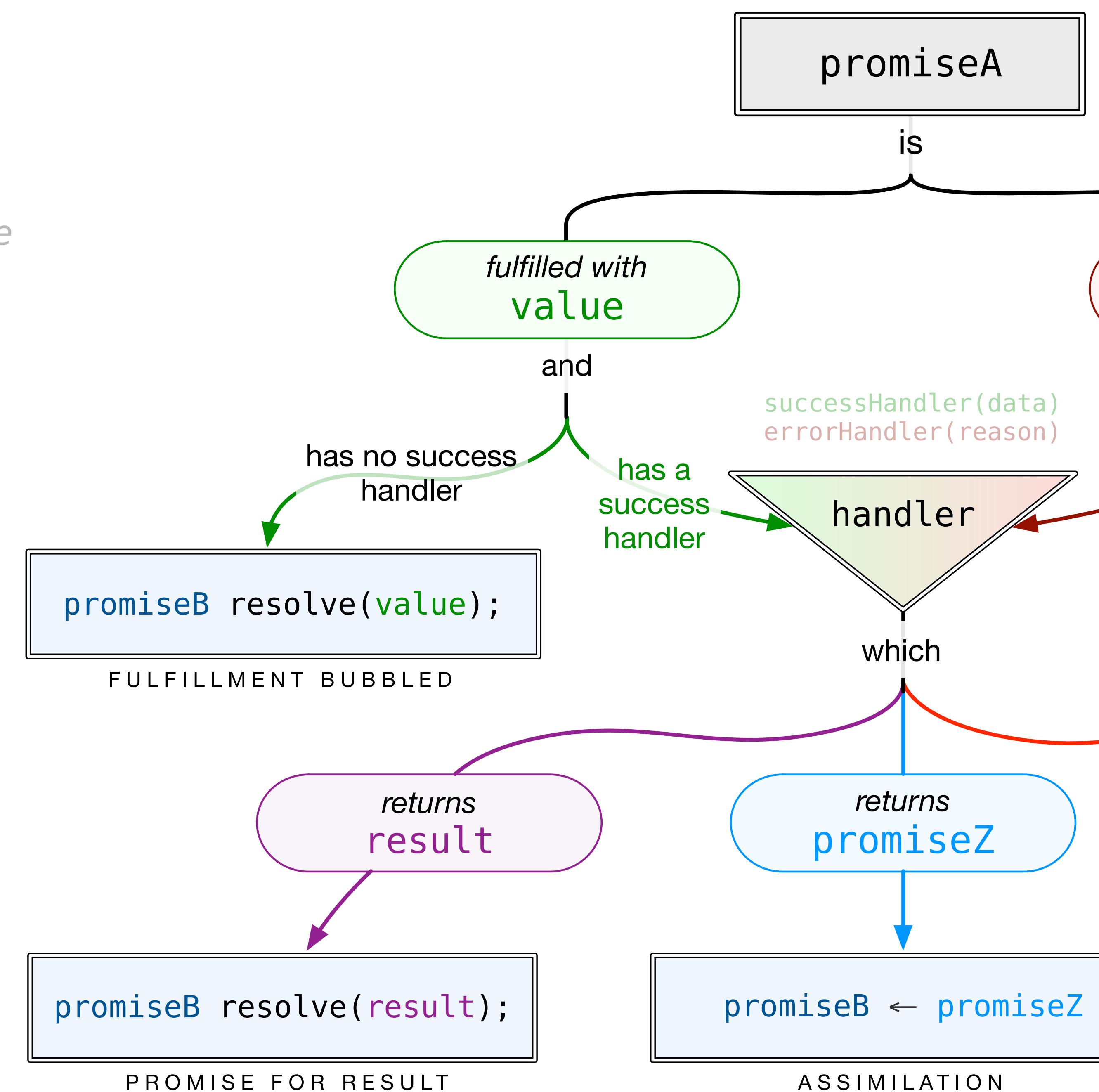


// output promise "becomes" returned promise

```
promiseForMessages = promiseForUser
  .then( function success (user) {
    // do some code to get a new promise
    return promiseForMessages;
});
```

// same idea, shown in a direct chain:

```
promiseForUser
  .then( function success (user) {
    // do some code to get a new promise
    return promiseForMessages;
})
  .then( function success (messages) {
    console.log( messages );
});
```





Review: Returning from Handler

```
// output promise is for returned val
```

```
promiseForVal2 = promiseForVal1
  .then( function success (val1) {
    val2 = val1++;
    return val2;
});
```

```
// same idea, shown in a direct chain:
```

```
promiseForVal1
  .then( function success (val1) {
    // do some code to make val2
    return val2; 
  })
  .then( function success (val2) {
    console.log( val2 );
  });
});
```

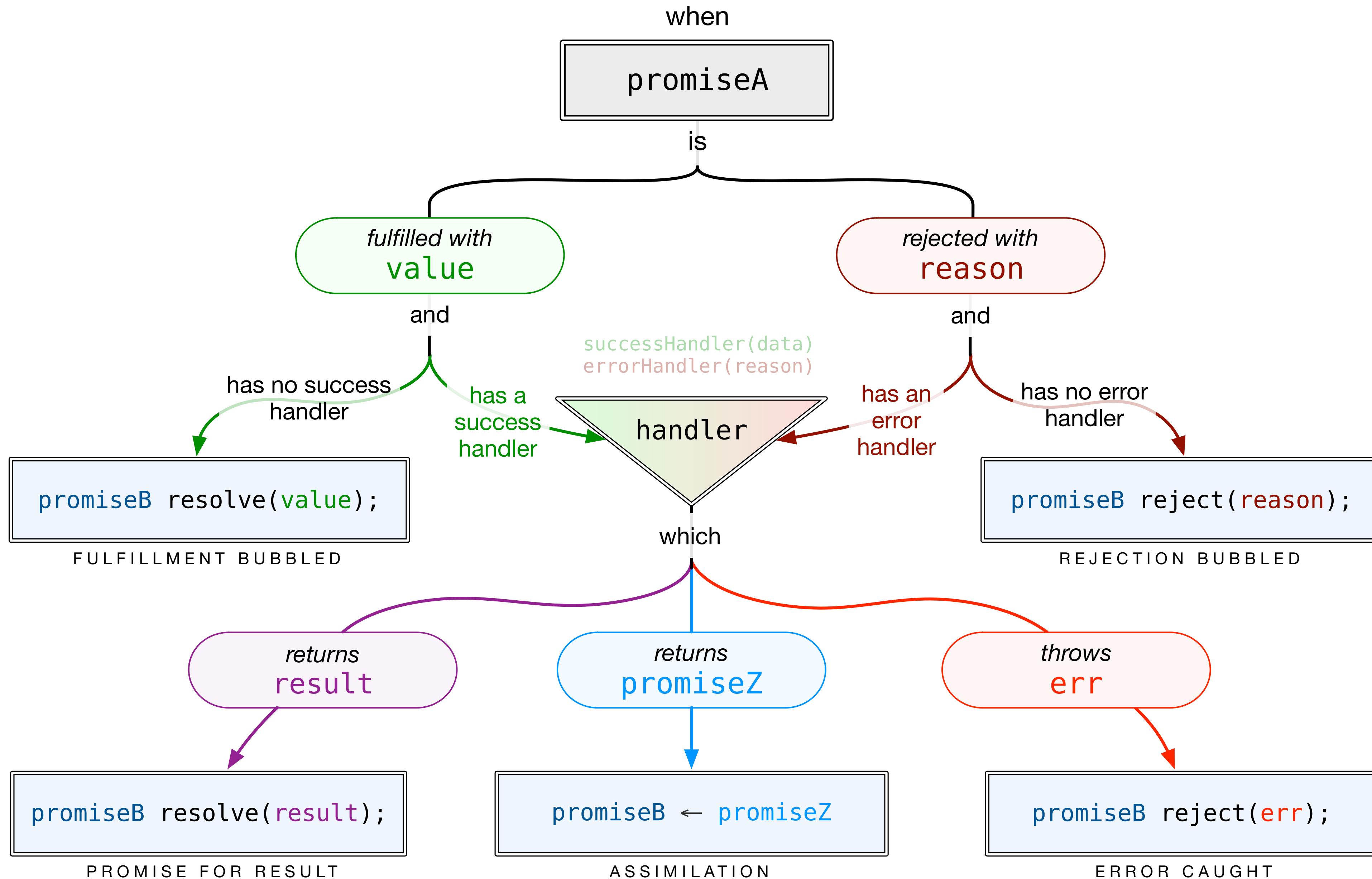
```
// output promise "becomes" returned promise
```

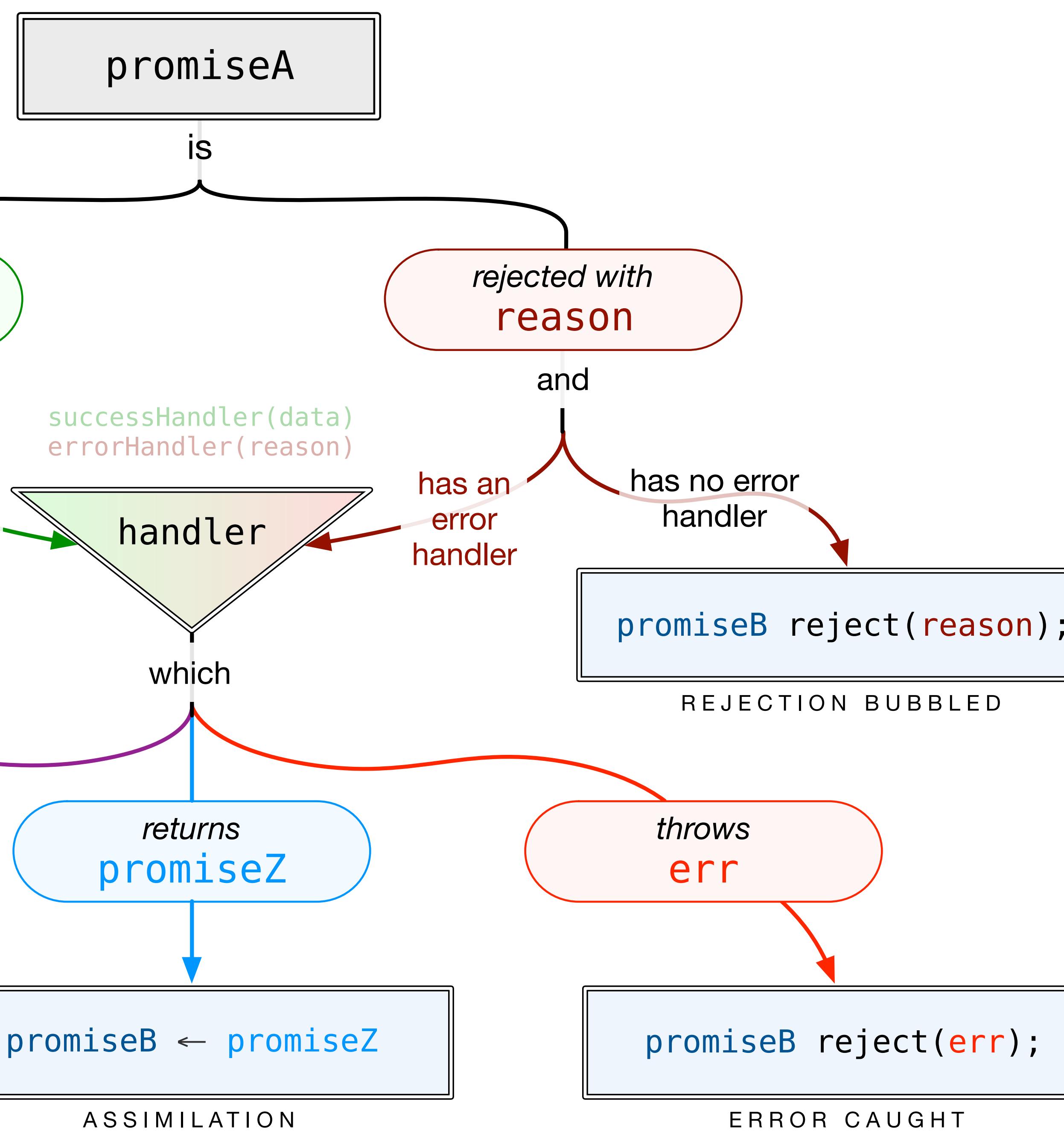
```
promiseForMessages = promiseForUser
  .then( function success (user) {
    // do some code to get a new promise
    return promiseForMessages;
});
```

```
// same idea, shown in a direct chain:
```

```
promiseForUser
  .then( function success (user) {
    // do some code to get a new promise
    return promiseForMessages;
})
  .then( function success (messages) {
    console.log( messages );
});
```

```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```





```

// output promise will be rejected with error

promiseForVal2 = promiseForVal1
  .then( function success (val1) {
    // THROWN ERROR '404' trying to make val2
    return val2;
  });

// same idea, shown in a direct chain:

promiseForVal1
  .then( function success (val1) {
    // THROWN ERROR '404' trying to make val2
    return val2;
  })
  .then( null, function failed (err) {
    console.log('Oops!', err);
  });
  
```

Danger: Silent Errors

```
myPromise
  .then(function (data) {
    use(data);
  })
  .catch(function (err) {
    doSomethingRiskyWith(err);
  })
  .done();
```

- ➊ Since `.then` (also `.catch`) always returns a new promise, it never throws an error, but instead rejects the outgoing promise
- ➋ Therefore, end promise chains in a `.done` or `.finally`, which do not export a new promise; any errors will be thrown as normal

that was .then

```
// array of API calls to make
var apiCalls = [
  '/api1/',
  '/api2/',
  '/api3/'
];
// map each url to a promise for its call result
apiCallPromises = apiCalls.map( function makeCall (url) {
  return $http.get(url).then( function got (response) {
    return response.data;
  });
});
// make a promise for an array of results once all arrive:
var thingsPromise = $q.all( apiCallPromises );
// use it:
thingsPromise.then( function got (results) {
  results.forEach( function print (result) {
    console.log(result);
  });
});
```

Extra: MEAN Promises



Node.js promise libraries: Q & Bluebird

```
npm install q --save
```

```
var Q = require('q');
```

```
npm install bluebird --save
```

```
var bluebird = require('bluebird');
```



Mongo (via Mongoose) Promises

```
var userPromise = User.find({ age: 30 }).exec();
```

```
var userPromise = User.create({ name: 'Gandalf' });
```

```
var userPromise = User.populate('friends');
```



Example: Mongoose `findOne`

```
function findOrCreate(model, properties) {
  return model.findOne(properties).exec().then(function(instance){
    if (instance) return instance; // --> promise for found instance
    return model.create(properties); // --> promise for created instance
  });
}

findOrCreate(User, {role: 'president'}).then(
  function (president) { res.json(president); }
).catch(
  function (err) { res.status(404).json(err); }
);
```

AngularJS

- **Uses many promises internally, via the \$q service (based on Q)**

```
var promiseForDate = $timeout(function(){  
    return +new Date();  
}, 1000);
```

```
var promiseForUser = $http.get('/users/1');
```

- **You can inject \$q to use its .all function or make new promises**

```
angular.module('myApp').service('myService', function ($q) { ... });
```

AngularJS \$http: use .then, not .success

<https://github.com/angular/angular.js/issues/10508>

```
$http.get('/users/1')
  .success(function(data, status, heads, cfg) {
    // use data
  })
  .error(function(data, status, heads, cfg) {
    // handle it
});
```

```
$http.get('/users/1')
  .then(function(response) {
    // use response.data
  })
  .catch(function(err) {
    // handle err
});
```

Why? Because .success & .error both return the ORIGINAL PROMISE,
not a new promise based on return value!



External Resources for Further Reading

- [AngularJS documentation for \\$q](#)
- [Kris Kowal & Domenic Denicola: Q](#) (the library \$q mimics; great examples & resources)
- [The Promises/A+ Standard](#) (with use patterns and an example implementation)
- [HTML5 Rocks: Promises](#) (deep walkthrough with use patterns)
- [Xebia: Promises and Design Patterns in AngularJS](#)
- [AngularJS Corner: Using promises and \\$q to handle asynchronous calls](#)
- [DailyJS: Javascript Promises in Wicked Detail](#) (build an ES6-style implementation)
- [MDN: ES6 Promises](#) (upcoming native functions)
- [Promise Nuggets](#) (use patterns)
- [Promise Anti-Patterns](#)
- [AngularJS / UI Router / Resolve](#)