Mechanics of Promises

Understanding JavaScript Promise Generation & Behavior

Async: continuation-passing

```
// Express & (old/imaginary) Sequelize
Page.findOne({where: {name: 'Promises'}}, function(err, page){
  if (err) return res.status(500).end();
  res.json(page);
});
```

Async: promise

```
// Express & (new) Sequelize!
Page.findOne({where: {name: 'Promises'}}).then(
  function (page) { res.json(page); },
  function (err)  { res.status(500).end(); }
);
```

Why?

Async: promise

```
// Express & Sequelize
Page.findOne({where: {name: 'Promises'}}).then(
  function (page) { res.json(page); },
  function (err)  { res.status(500).end(); }
);
```

Break free from the async call!

```
var pagePromise = Page.findOne({where: {name: 'Promises'}});

// promise is portable – can move it around
pagePromise.then(
  function (page) { res.json(page); },
  function (err)  { res.status(500).end(); }
);
```

Export to other modules…

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

make a cache out of them

```
// cache: {[url: String]: Promise<Blob>}
const cache = {}

// get(url: String) -> Promise<Blob>
function get(src) {
  if (!cache[src])
   cache[src] = fetch(src).then(res => res.blob())
  return cache[src]
}

get(‘http://example.com/some.data')
  .then(data => show(data))
```

…collect in arrays and pass into functions…

```
var dayPromises = [];

// make 7 parallel (simultaneous) day requests
for (var i = 0; i < 7; i++) {
  var promiseForDayI = Day.findOne({where: {dayNum: i}});
  dayPromises.push(promiseForDayI);
}

// act only when they have all resolved
Promise
  .all(dayPromises)
  .then(days => res.render('calendar', {days}))
```

…and much more

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

So, what is a promise?

magic!
(no)
iHDWALLPAPERS.COM

Promises are Objects

state (pending, fulfilled, or rejected)

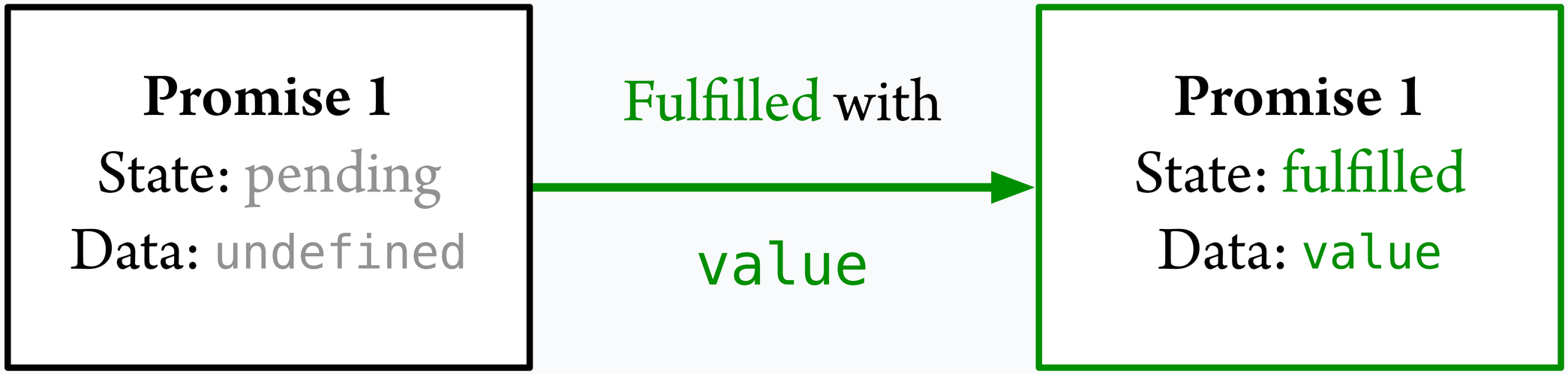
information (value or a reason)

} (hidden if possible)

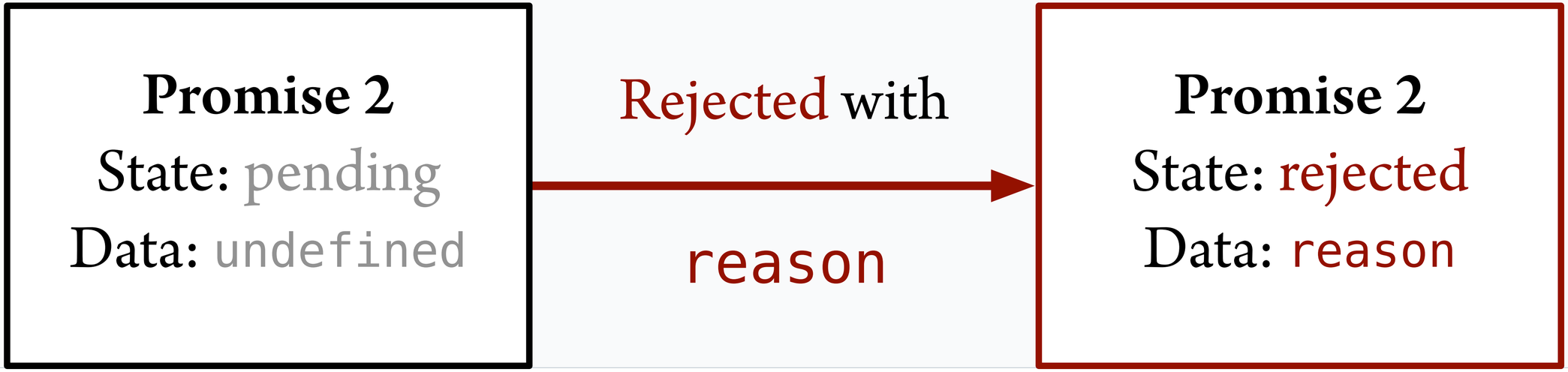
`.then()`

`.catch()`

} (public interface)



promises only change state while pending



Promise 1
State: pending
Data: `undefined`

Fulfilled with

`value`

Promise 1
State: fulfilled
Data: `value`

```
aPromise.then( successHandler, failureHandler )
```

Promise 2
State: pending
Data: `undefined`

Rejected with

`reason`

Promise 2
State: rejected
Data: `reason`

Timing-ambivalent

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

- 1. *Add handler*
  2. promise settles
  3. handler is called once

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

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

What does this solve?

Callback Hell

```
// 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(user => get(user.messageIDs))
  .then(messages => get(messages[0].commentIDs))
  .then(comments => UI.show( function (comments) {
    UI.display( comments[0] );
  })
  .catch( function (err) {
    console.log('Fetch error: ', err);
  })
```

Black Holes

```
// 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];
});

var headline = person.name; // 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

A+

- The promises standard · Promises/A+
  - ES6 promises work natively in most browsers
  - Created with the *new Promise* constructor

```
new Promise((resolve, reject) => resolve(10))
new Promise((resolve, reject) => reject(new Error('ahh!'))
```

So where do real promises come from?

- Existing libraries may return promises
  - `fetch` in the browser
  - Sequelize queries / db actions
- Wrap async calls in promise-makers
  - ES6 / `Bluebird` *constructor*
- Promise libraries can wrap for us, e.g. in Node
  - `Bluebird.promisifyAll(fs)`

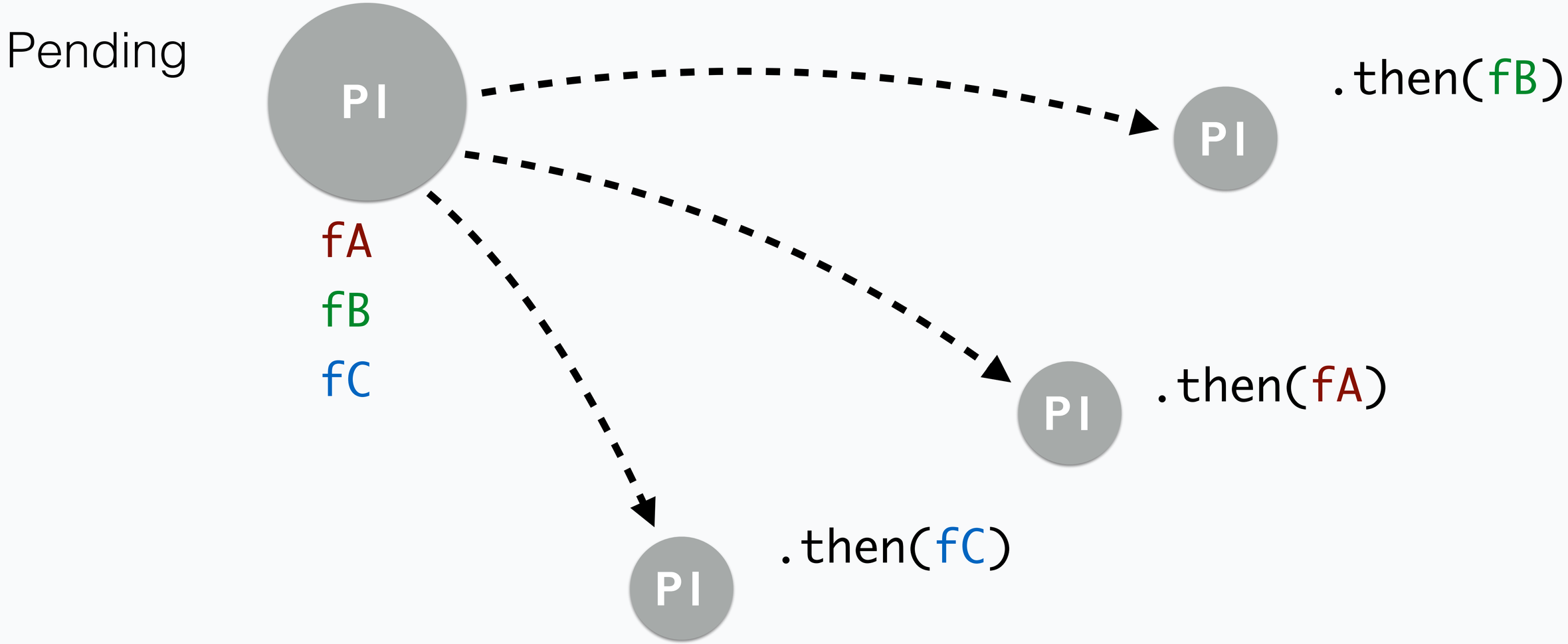
Promisification in Node.js

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

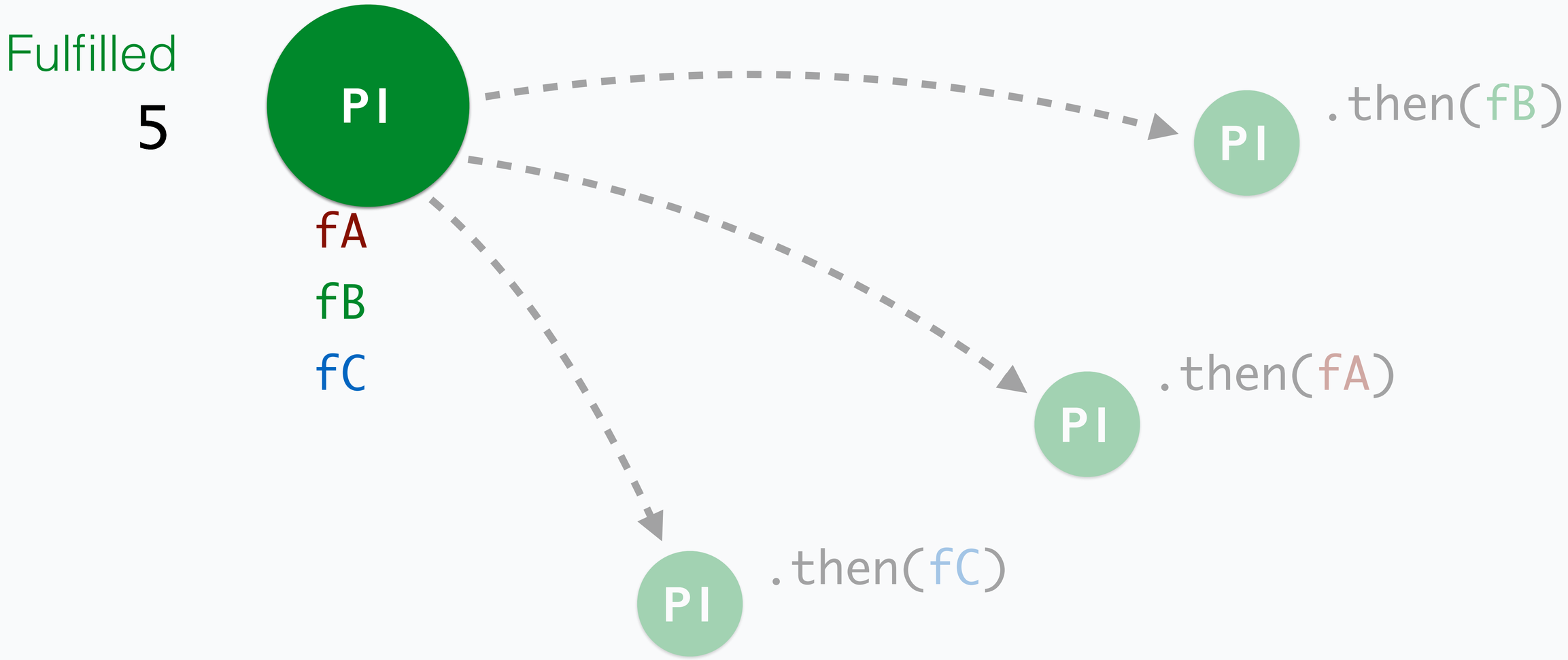
```
Bluebird.promisifyAll( fs );
fs.readFileAsync('file.j', ‘utf8').then(function (text) {
  // use the text
});
```

(Break)

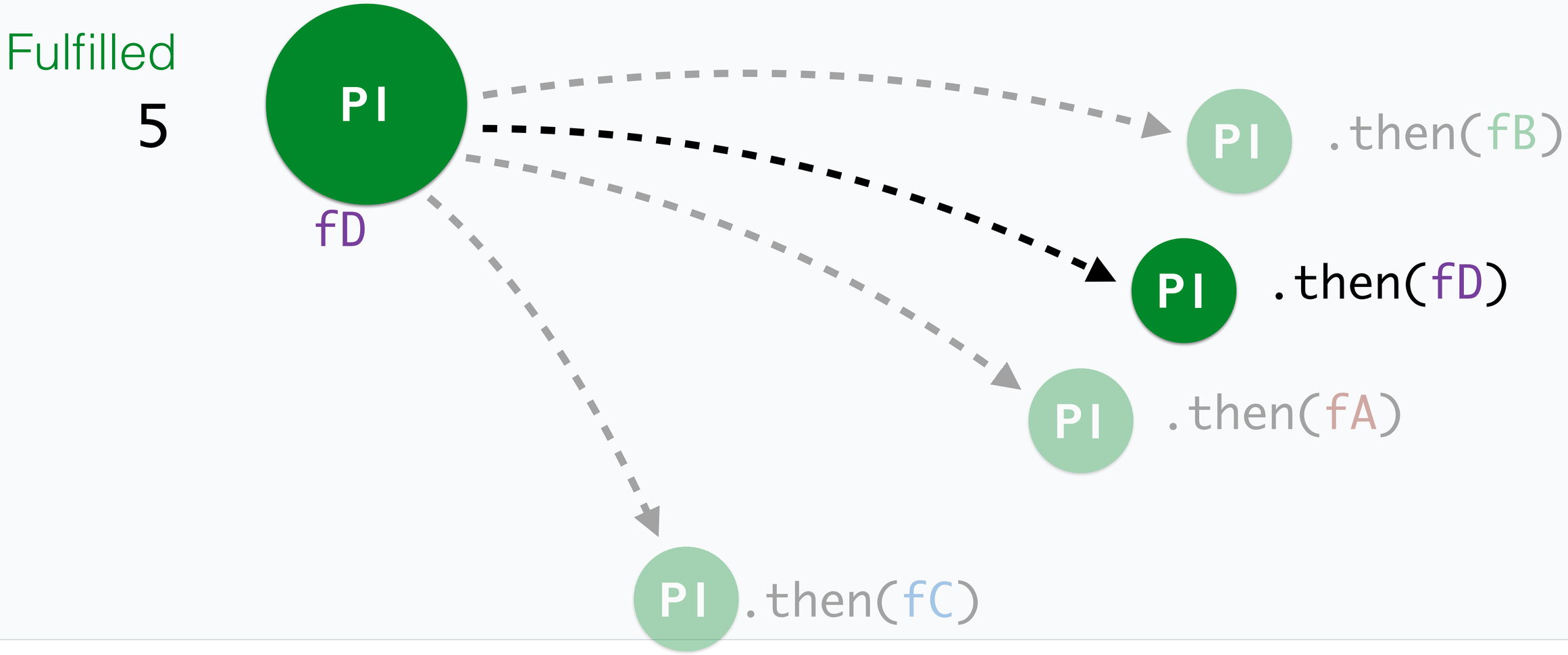
.then on same promise (not chaining!)



.then on same promise (not chaining!)



.then on same promise (not chaining!)



fluent magic: `.then` and `.catch` return Promises

```
interface Promise {
  then(onFulfilled: (any)->any?) -> Promise
  catch(onRejected: (any)->any?) -> Promise
}
```

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

This is why we can chain .then

P1 `promiseForThing` P2

→ `.then(doStuff)` P3

→ `.then(doOtherStuff)` P4

→ `.then(doMoreStuff)` P5

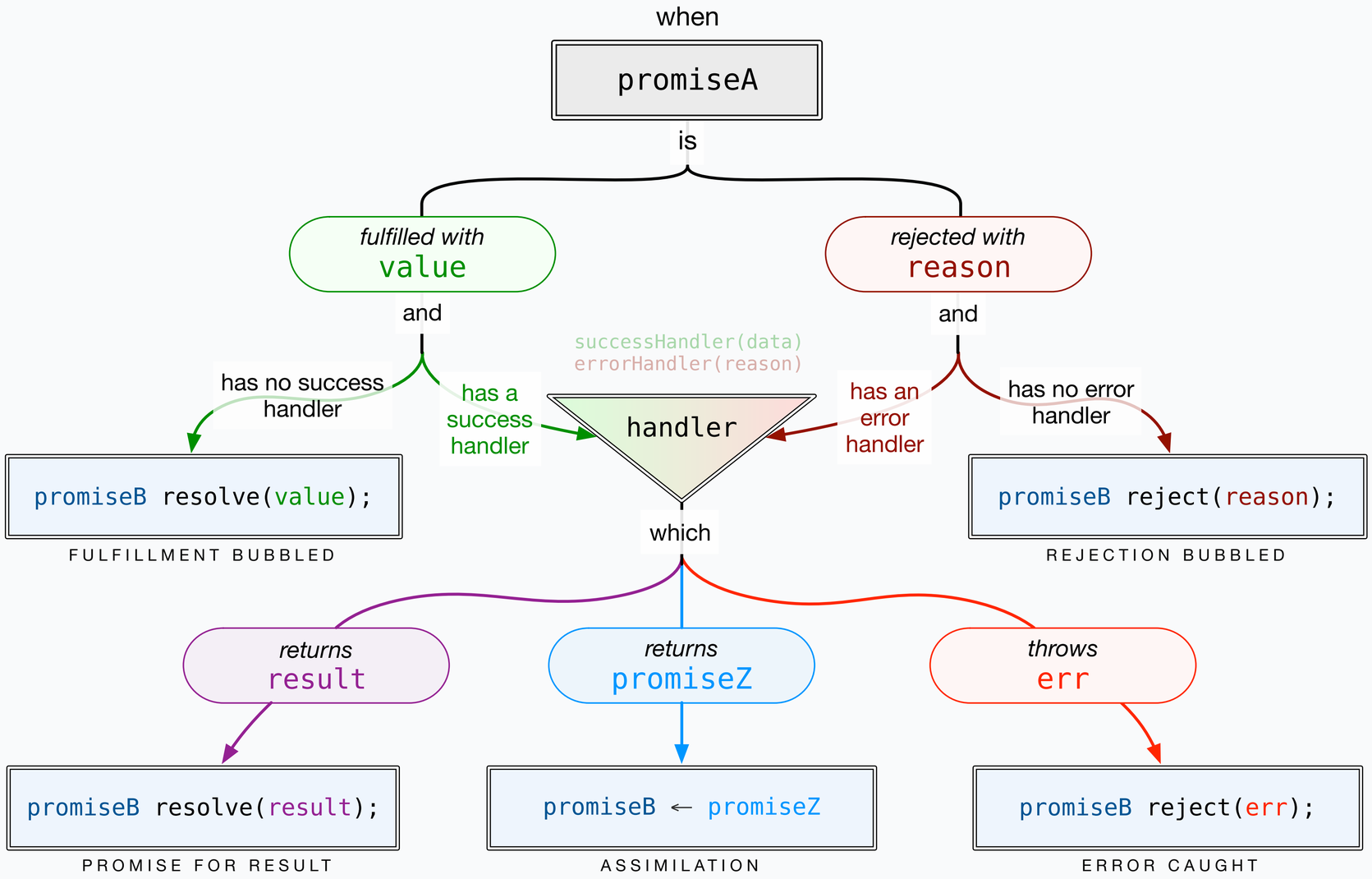
```
.catch(handleErr);
```

What is promiseB a Promise of?

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

Brace yourselves…

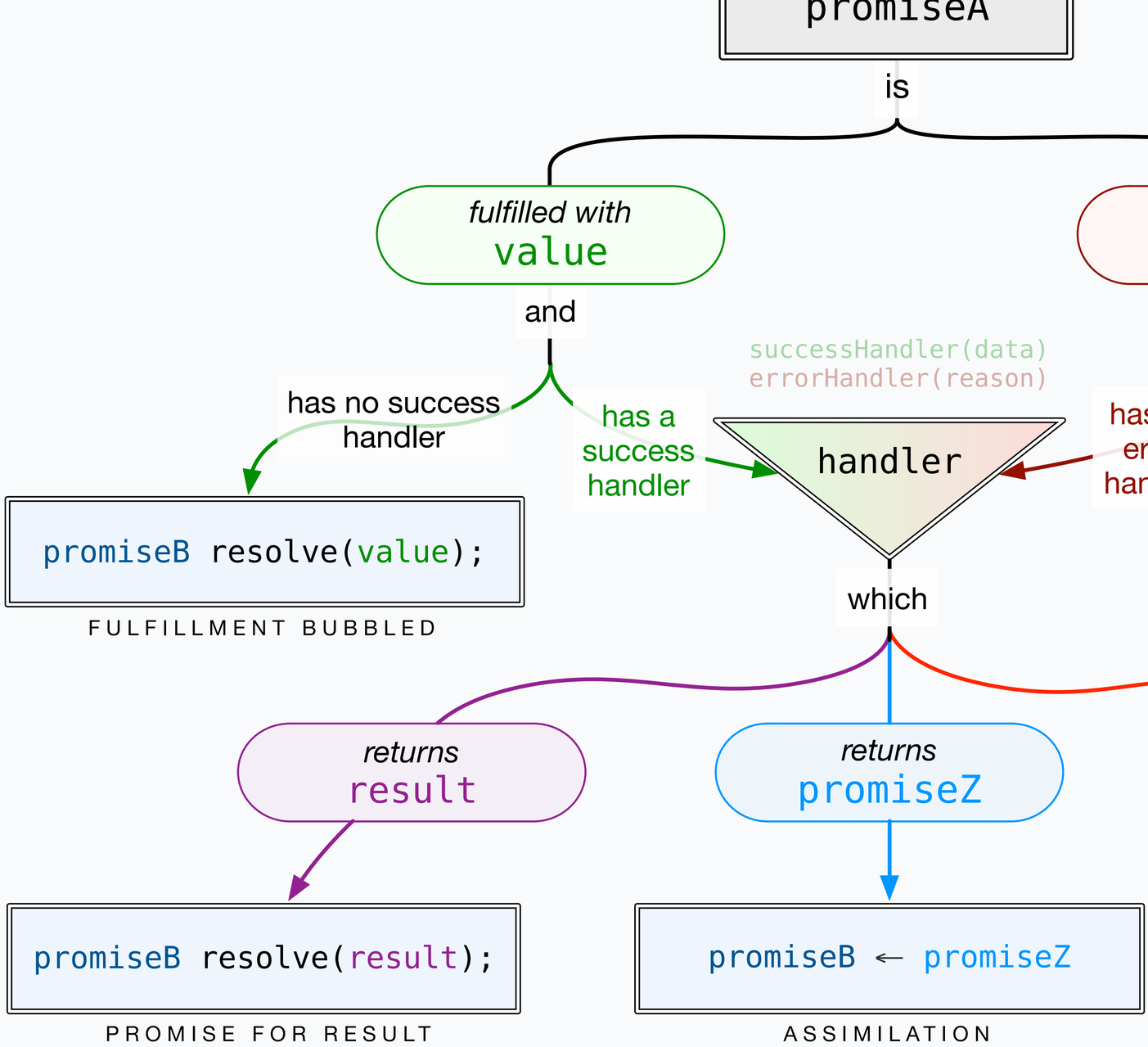
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



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

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

Fulfillment bubbled down to first available success handler:

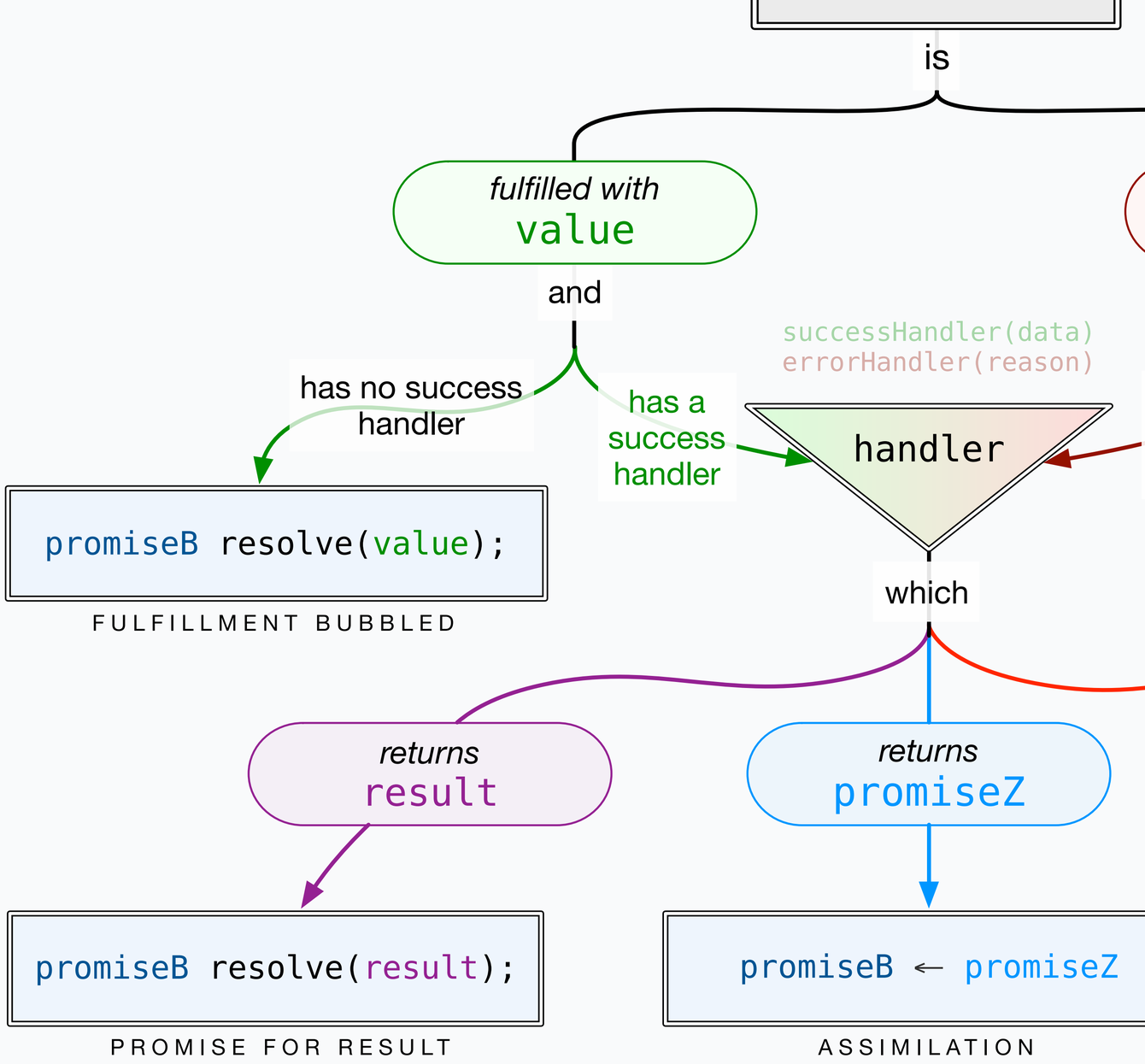
Console log reads "Hello."

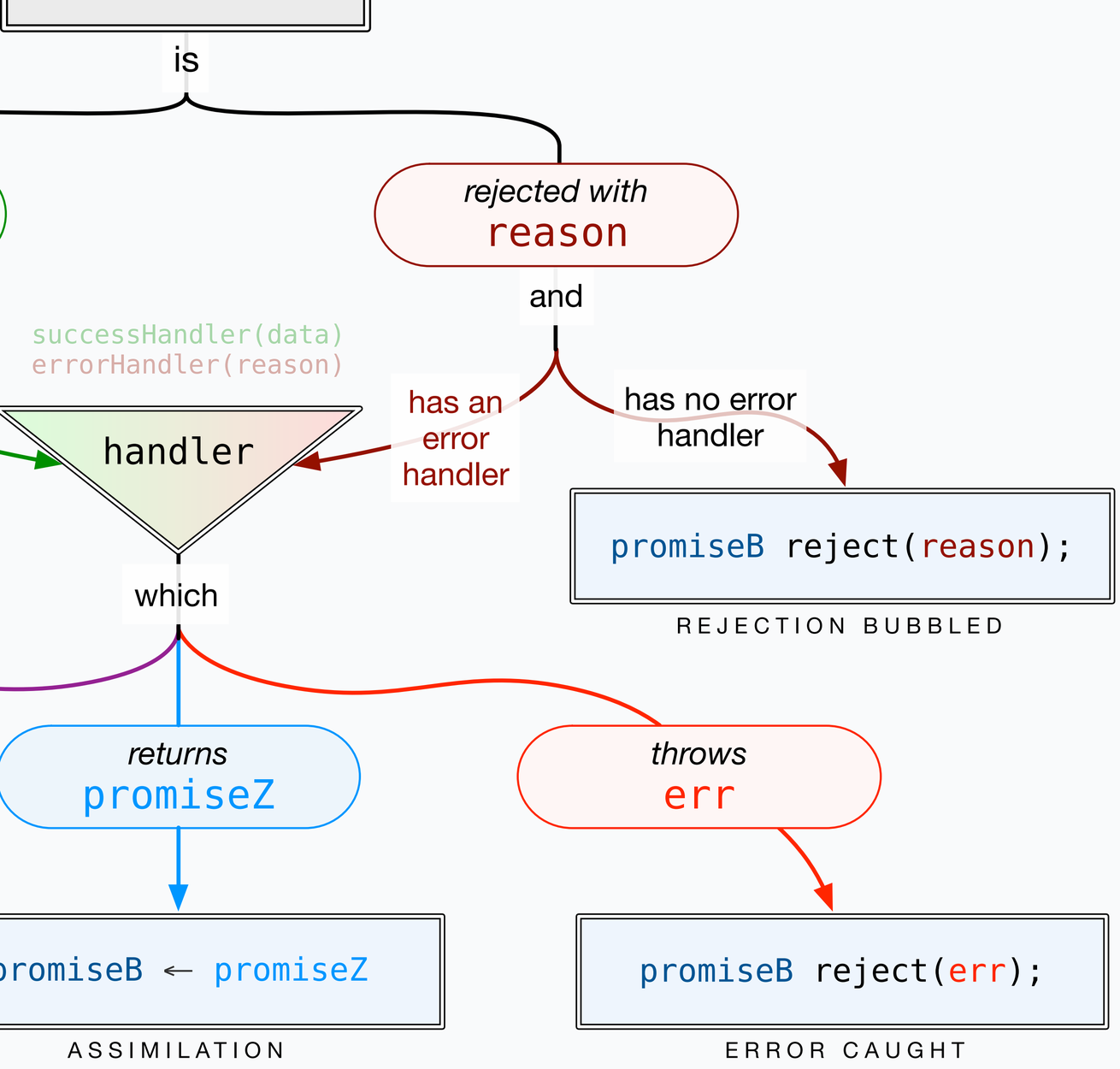


```
// promise0 fulfills with 'Hello.'

promise0
  .then(null, warnUser) // -> p1
  .then() // -> p2
  .then() // -> p3
  .then(null, null) // -> 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.”

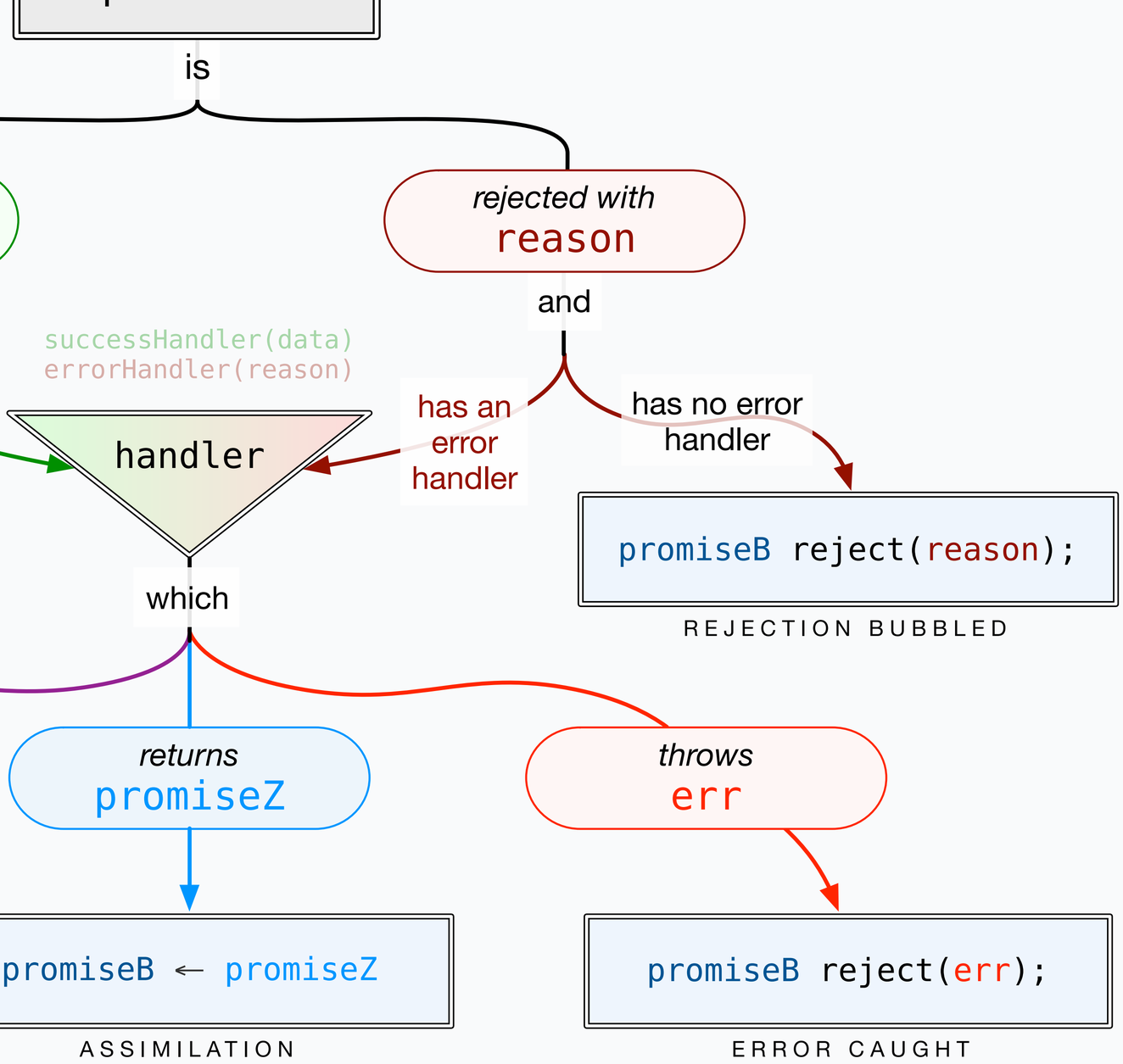





```
function logYell (input) {
  console.log(input+'!');
}

// promise0 rejected with 'Sorry'

promise0
  .then() // -> p1
  .then() // -> p2 and so on
  .then()
  .catch(logYell)
```

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

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

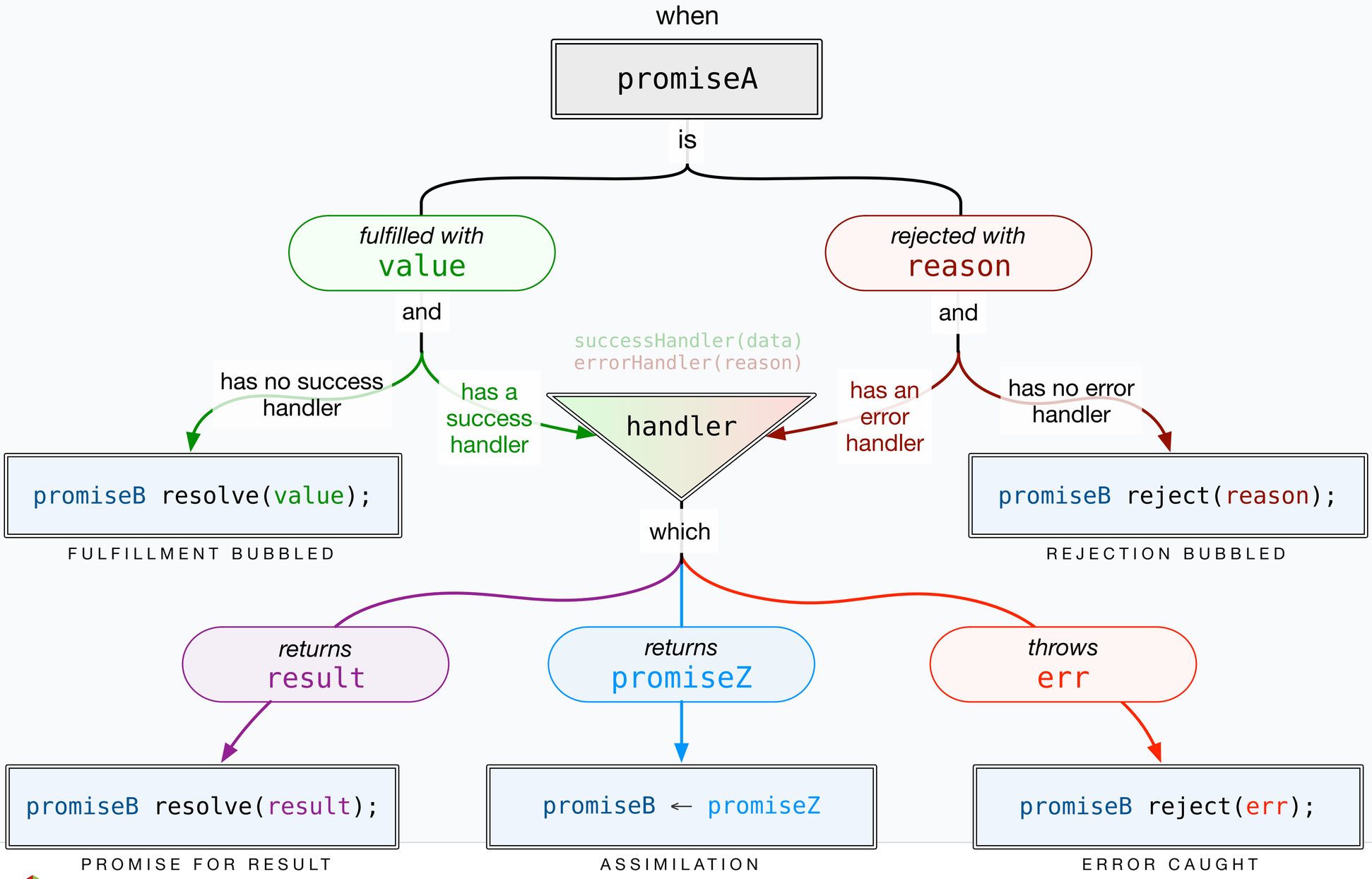
Console log is "Sorry!"

Review: Success & Error Bubbling

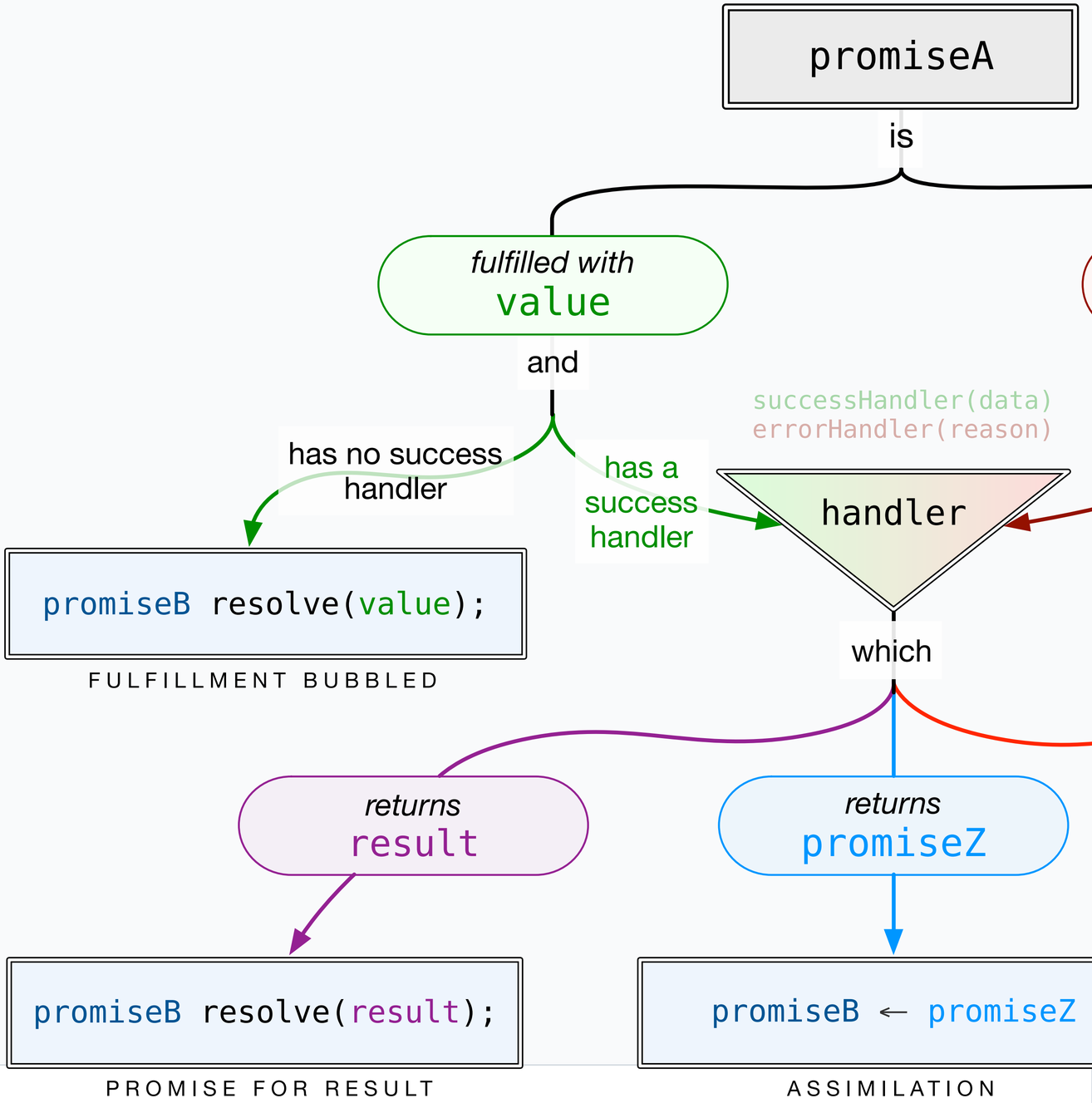
```
// promiseA is fulfilled with 'hello'

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
```

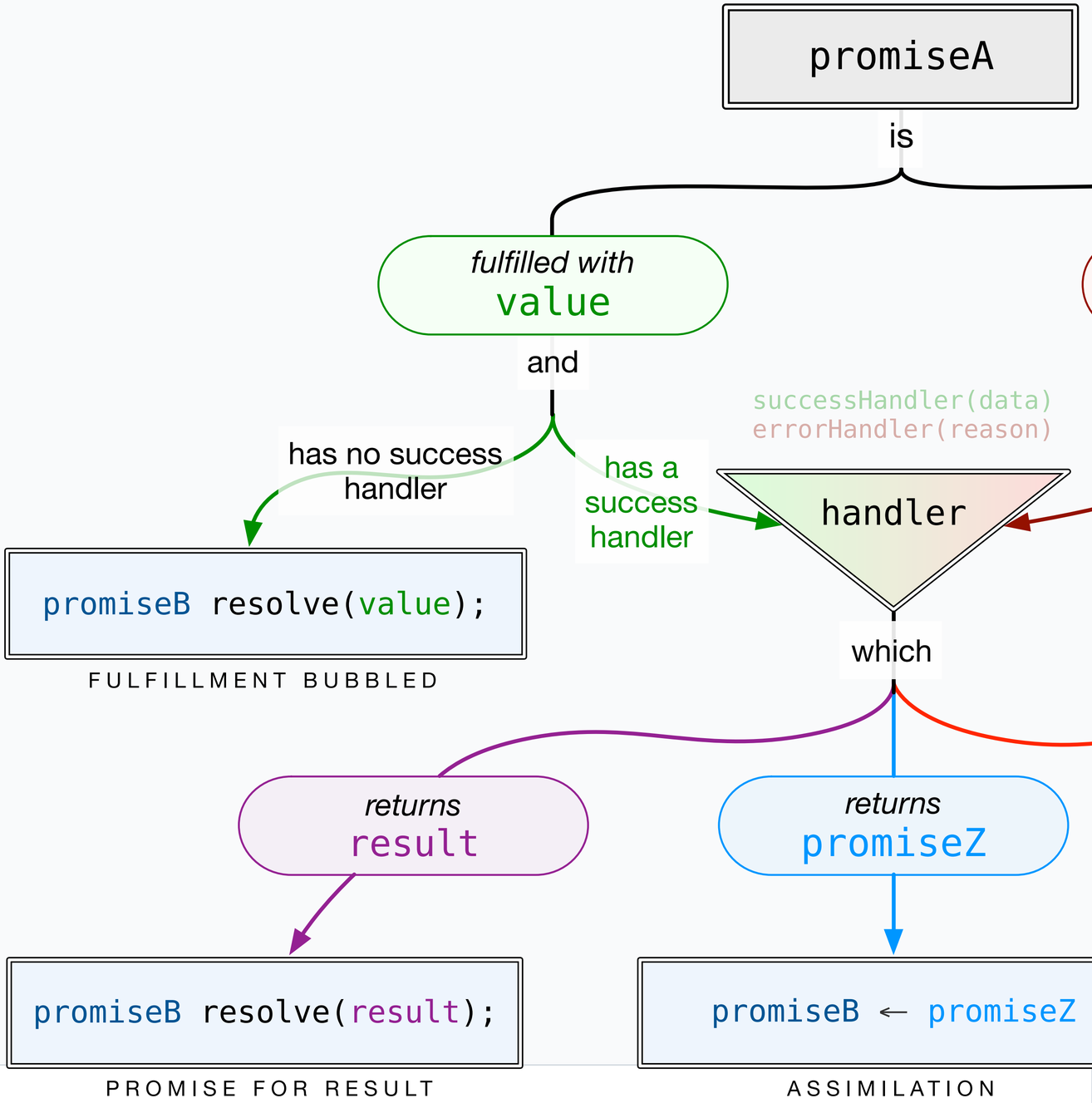
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



```
// 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 );
  });
```

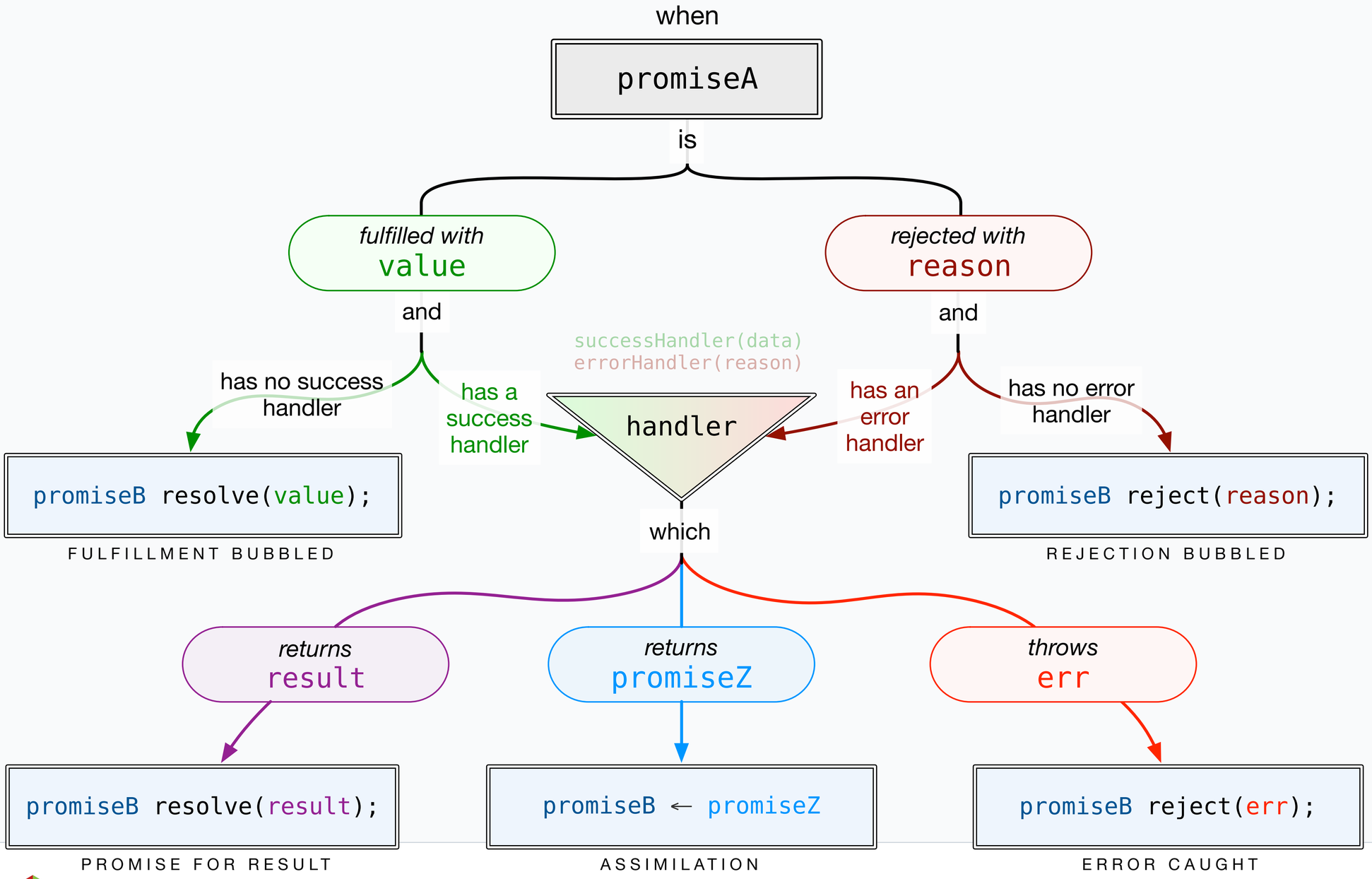


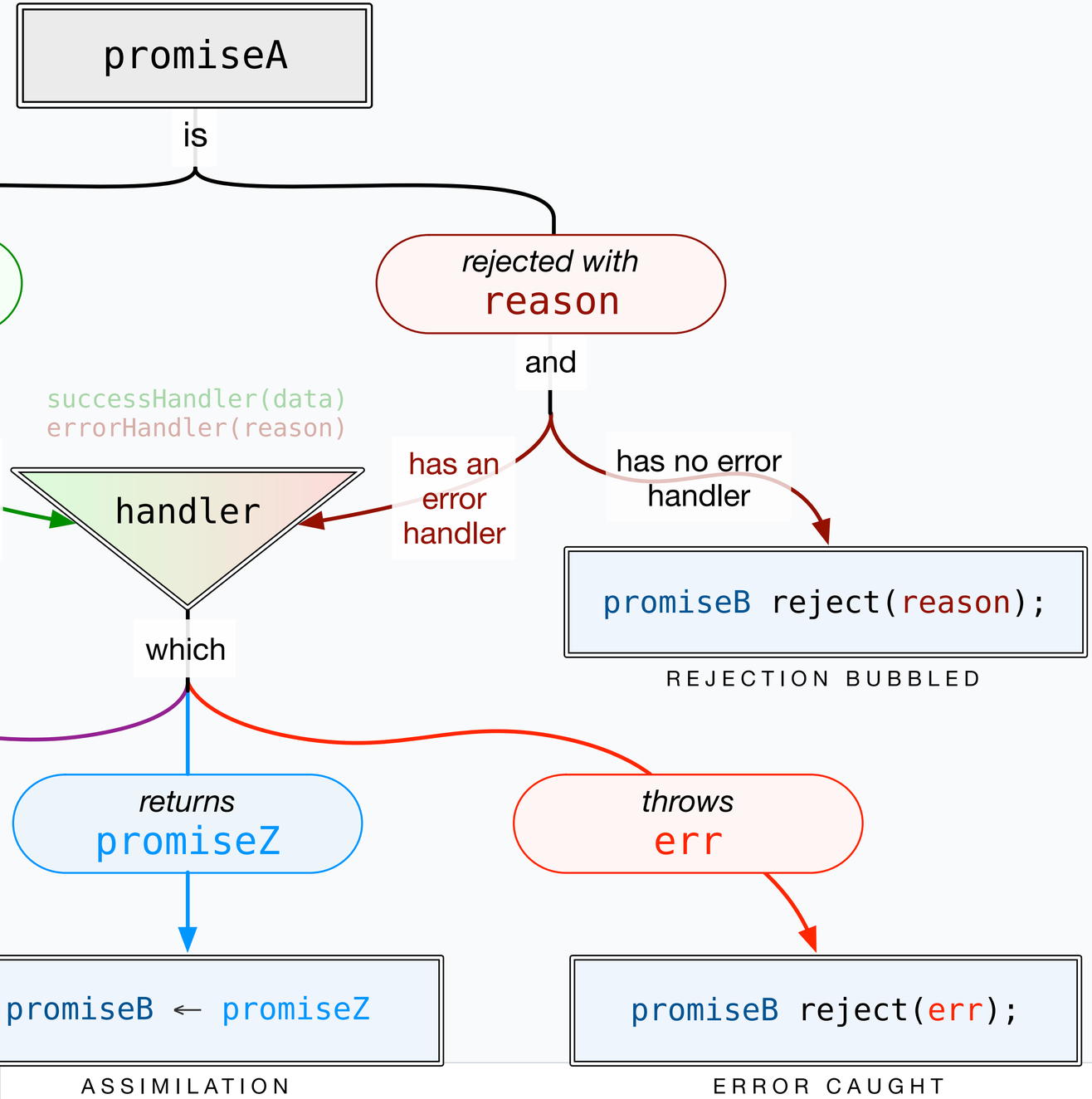
Review: Returning from Handler

```
// 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);
  });
```

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);
  });
});
```

~~MEAN~~ SEAN(?) Promises

Node.js promise libraries: Q & Bluebird

```
npm install q --save

const Promise = require('q');
```

```
npm install bluebird --save

const Promise = require('bluebird');
```

You don't need either of them—Node comes with promises.
But you may want some extra goodies they give you:
`Promise.props`, `Promise.spread`, etc.

(some) Sequelize Promises

```
var usersPromise = User.find({where: {age: 30}});

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

var savedUserPromise = gandalf.save();

var userIsDestroyedPromise = gandalf.destroy();

var usersPromise = sequelize.query('SELECT * FROM users', {type:
sequelize.queryTypes.SELECT});

var syncedPromise = sequelize.sync();
```

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)
- We Have a Problem With Promises
- 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