Mechanics of Promises

Understanding JavaScript Promise Generation & Behavior

I Will Be Able To…

- Explain the behaviour of promises under multiple conditions (e.g. chaining on same promise, chaining without a handler, returning a new promise)
- Build my own promise library

Async: continuation-passing

```
// Express & node-postgres
client.query(‘SELECT * FROM tweets’, function (err, data){
  if (err) return next(err);
  res.json(data.rows);
});
```

Async: promise

```
// Express & Sequelize!
Page.findOne({where: {name: ‘Promises'}})
.then(function (page) {
  res.json(page);
});
```

Async: promise

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

Separate the async request from the eventual behavior we want to run

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

// promise is portable - can move it around
pagePromise.then(
  function (page) { res.json(page); },
  function (err)  { return next(err); }
);
```

Export to other modules…

```
const studentPromise = User.findOne({where: {role: 'student'}});
module.exports = studentPromise;
```

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

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

…and much more

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

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… with error handling, for extra hellishness

const userID = 'a72jd3720h';
getUserData( userID, function ( 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 ( 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);
  });
```

PROMISE ADVANTAGES

- Portable
- Multiple handlers
- “Linear” or “flat” chains
- Unified error handling

So, what is a promise?

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

— THE PROMISES/A+ SPEC

magic!

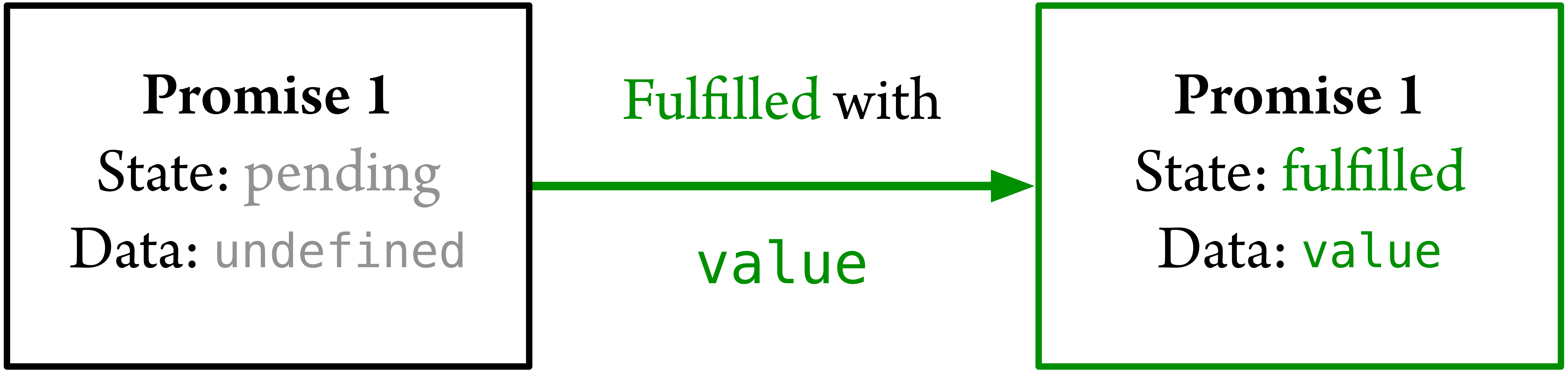
(no)

Promises are Objects

state (pending, fulfilled, or rejected) } (hidden if possible)

information (value or a reason)

`.then()` (public property)

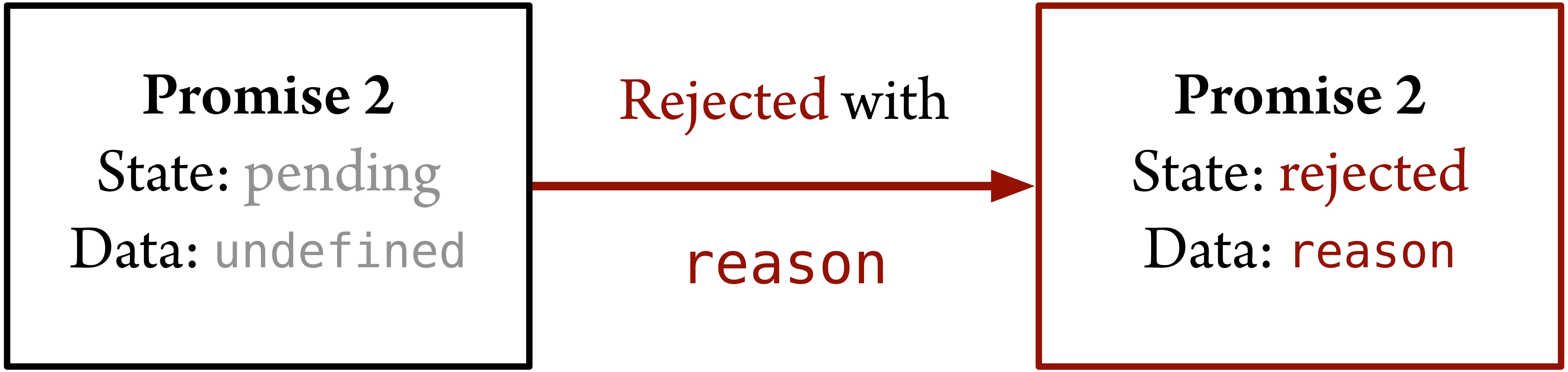


Promise 1
State: pending
Data: `undefined`

Fulfilled with
`value`

Promise 1
State: fulfilled
Data: `value`

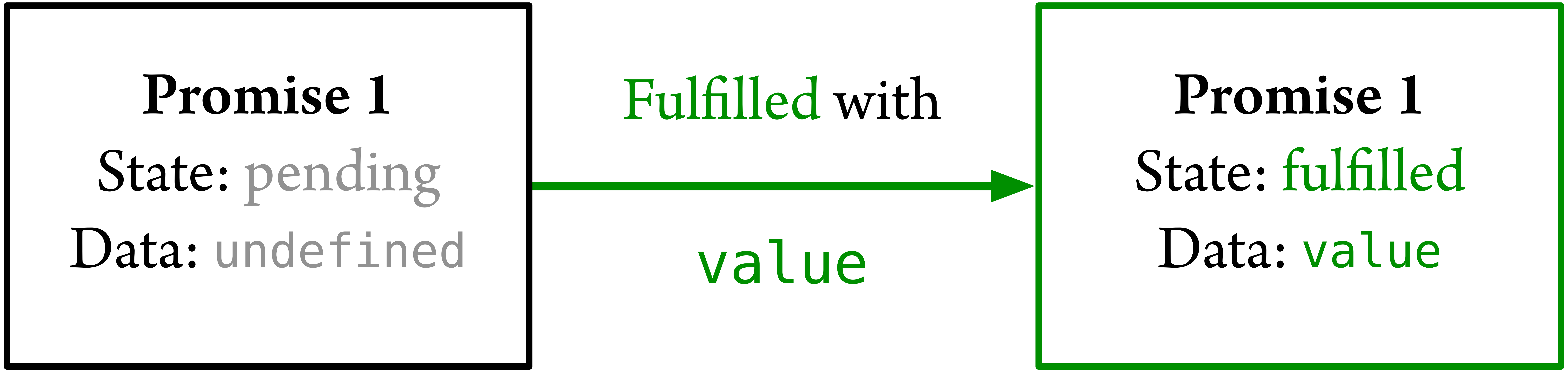
promises only change state while pending



Promise 2
State: pending
Data: `undefined`

Rejected with
`reason`

Promise 2
State: rejected
Data: `reason`



Promise 1

State: pending

Data: `undefined`

Fulfilled with

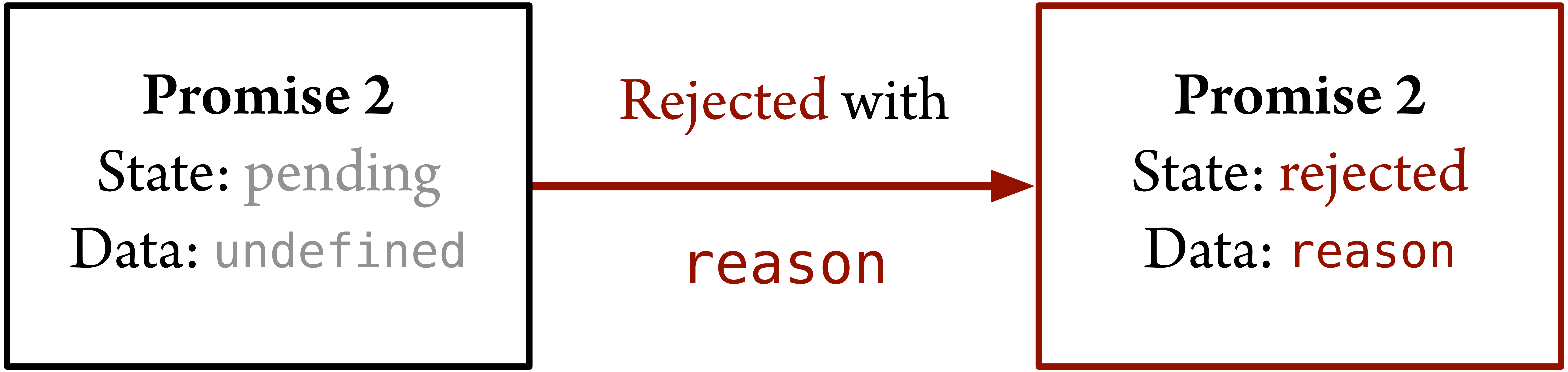
`value`

Promise 1

State: fulfilled

Data: `value`

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



Promise 2

State: pending

Data: `undefined`

Rejected with

`reason`

Promise 2

State: rejected

Data: `reason`

Standards

- **The standard which won: Promises/A+**
  - Only covers one function: `.then`!
- **ES6 promises are a superset of P/A+**
  - Includes some additional methods (`.catch`, `all`, `race`)
- **Main point: promises are *implemented*, not a fundamental type**
  - Some libraries followed earlier standards or no standard
  - Modern libraries follow Promises/A+

```
// Fantasy solution

const containerA = new Container();

asyncGetData( function ( data ) {
  containerA.save( data ); // once async completes
});

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

So where do real promises come from?

- Existing libraries may return promises
  - Sequelize queries / db actions
  - axios
  - fetch
- Wrap vanilla async calls in promise constructor
  - ES6 / `Bluebird`: `new Promise(executor)`
- Promise libraries can wrap for us, e.g. in Node
  - `Bluebird.promisifyAll(fs)`

Making New Promises: How?

new Promise(executor)

```
const promiseForTxt = new Promise( function (resolve, reject) {
  fs.readFile(‘log.txt', function (err, text) {
    if (err) reject( err );
    else resolve( text );
  });
});

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

Promisification in Node.js

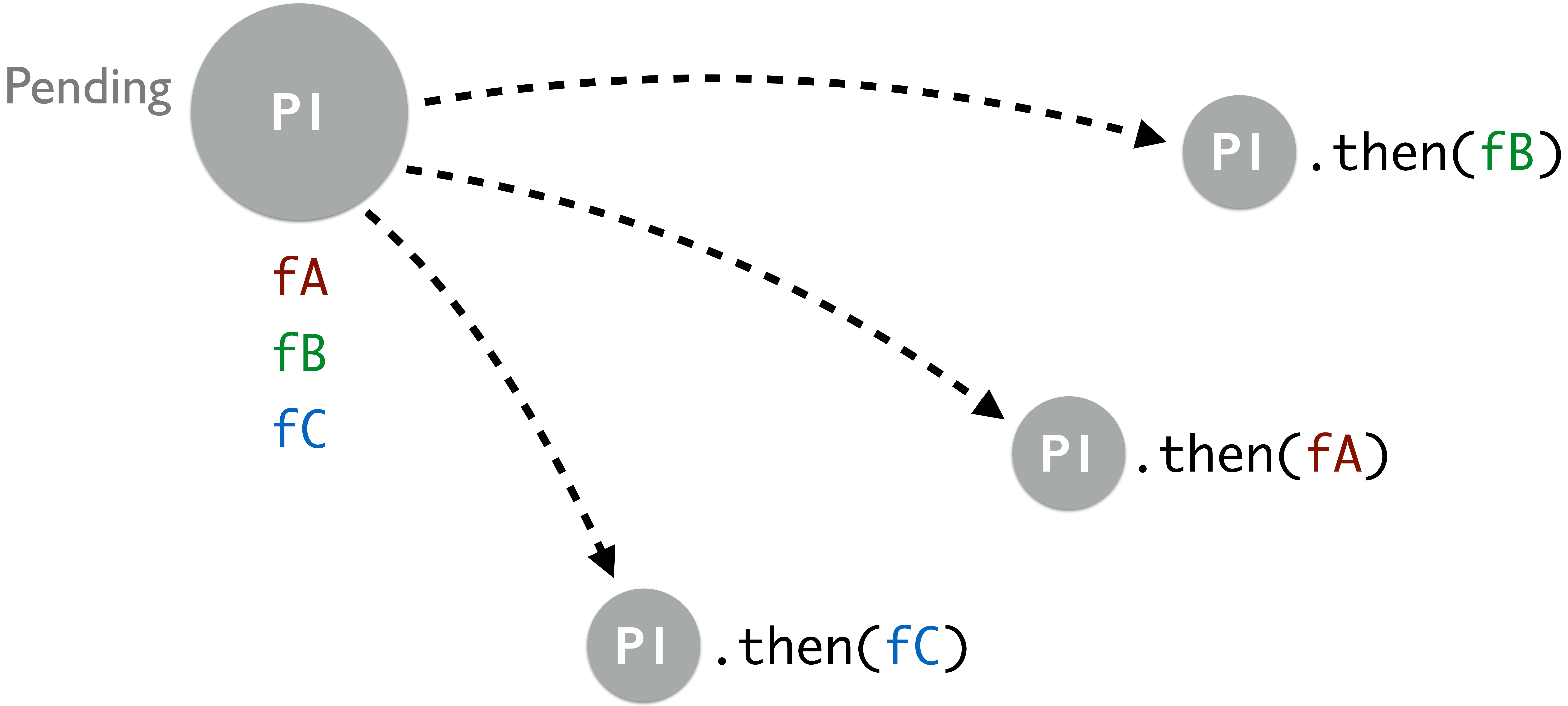
```
fs.readFile('foo.txt', 'utf-8', function (err, text) {
  // use the text
});
```

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

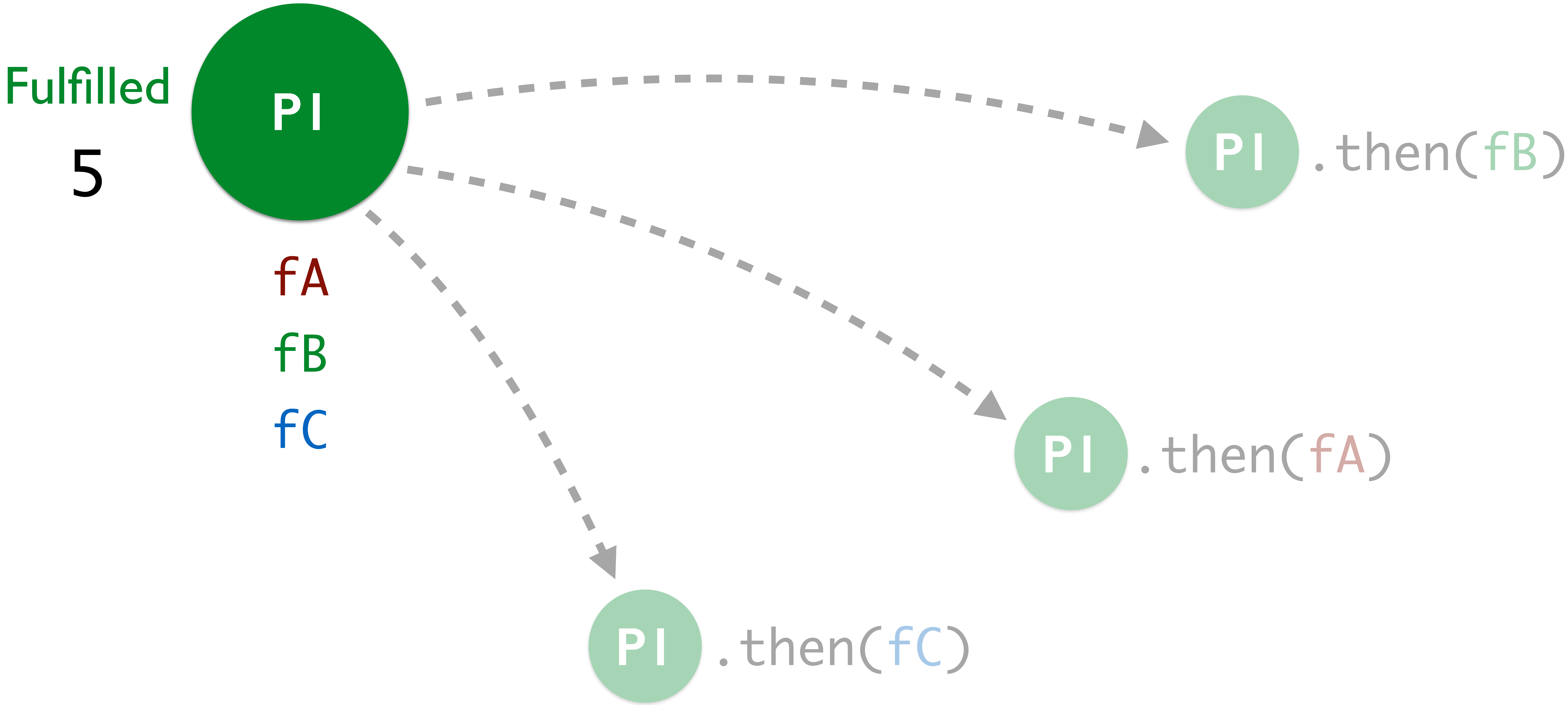
The Magic of Promises

- Magical behavior #1: It doesn’t really matter whether .then() is called before or after the promise is resolved. Everything just works properly!

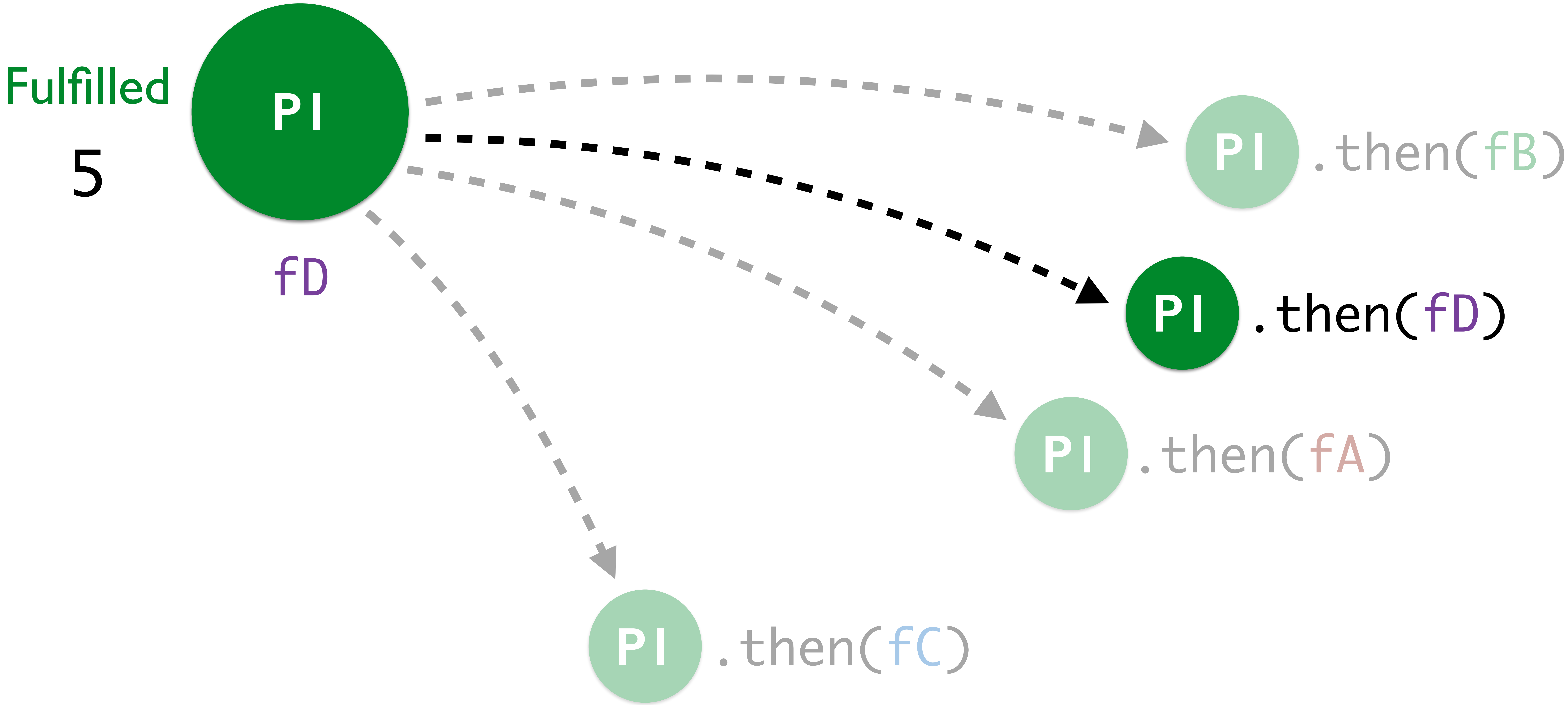
.then on same promise (not chaining!)



.then on same promise (not chaining!)



.then on same promise (not chaining!)



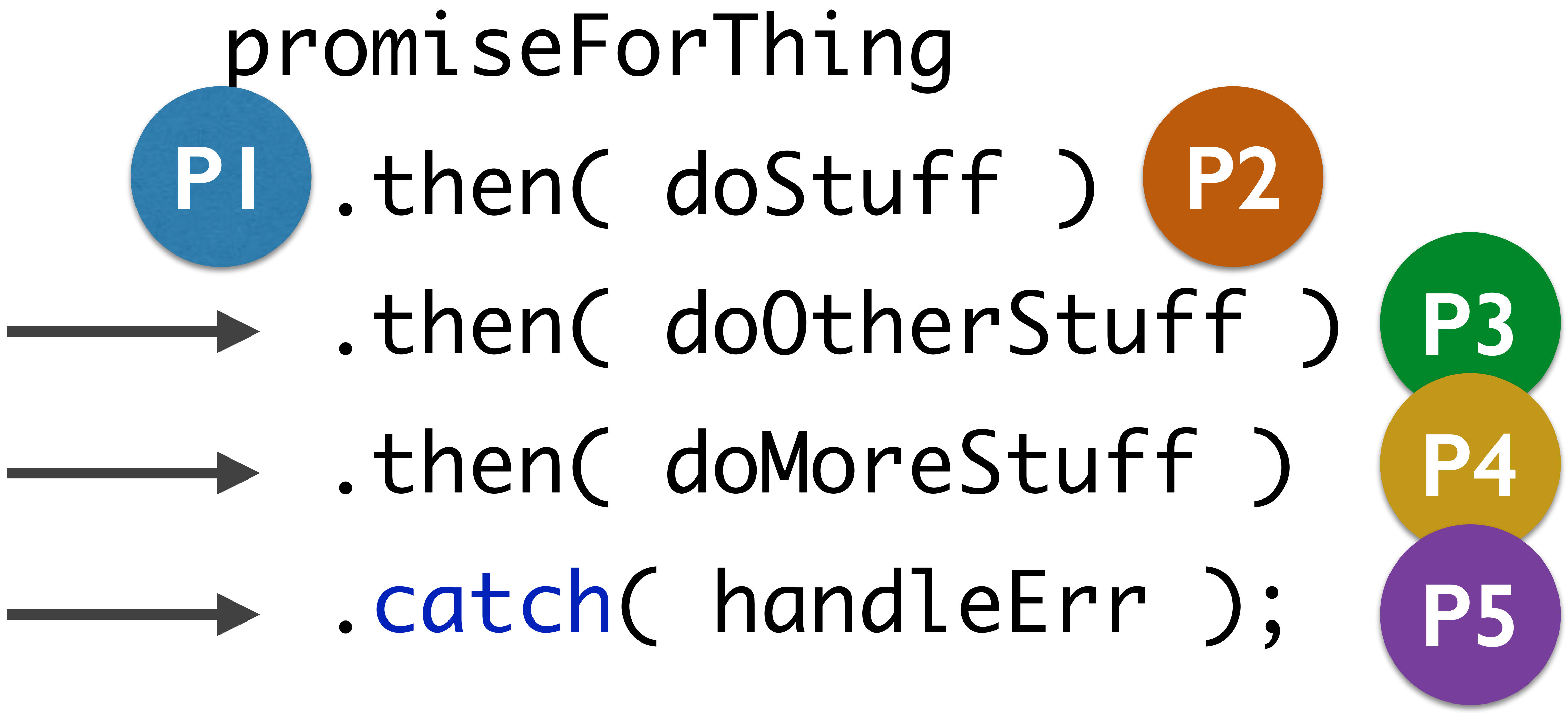
The Magic of Promises

- **Magical Property #1: It doesn’t really matter whether .then() is called before or after the promise is resolved. Everything just works properly!**
- **Magical Property #2: .then() returns a new (different) Promise.**
  - whatever the previous promise returned ends up as the resolved value in the new promise.
  - If the previous promise returned a promise… then the **resolved value** of the returned promise ends up in the new one. Woah.

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)`

So what happens if we ‘return’ in a handler

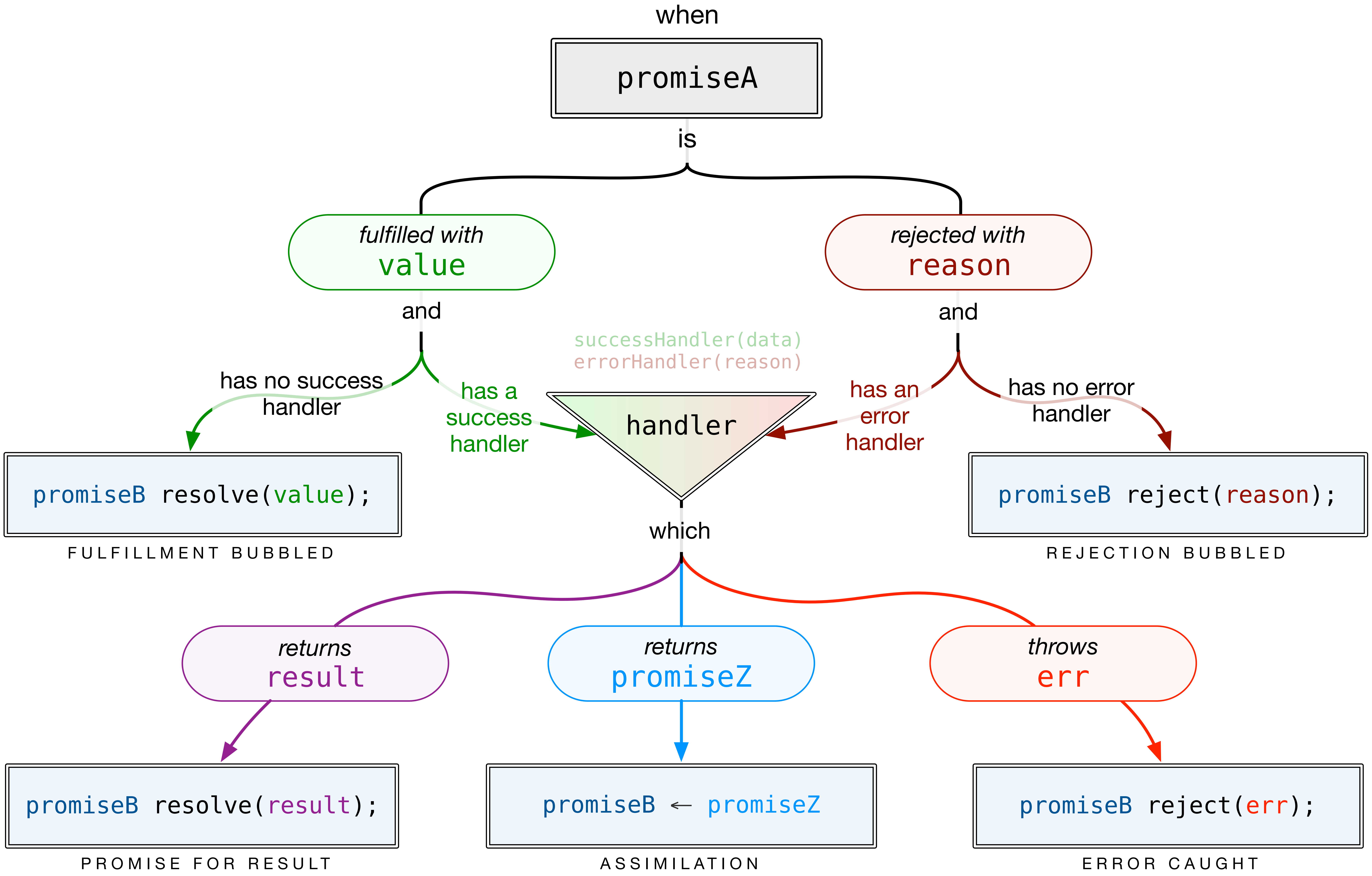
```
const promiseB = promiseA
  .then(function thingSuccess (thingA) {
    // run some code
    return thingB;
  })
```

Brace yourselves…

(Break)

Brace yourselves...

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



when

promiseA

is

fulfilled with value

and

has no success handler

promiseB resolve(value);

FULFILLMENT BUBBLED

has a success handler

successHandler(data)

errorHandler(reason)

handler

rejected with reason

and

has an error handler

has no error handler

promiseB reject(reason);

REJECTION BUBBLED

which

returns result

promiseB resolve(result);

PROMISE FOR RESULT

returns promiseZ

promiseB ← promiseZ

ASSIMILATION

throws err

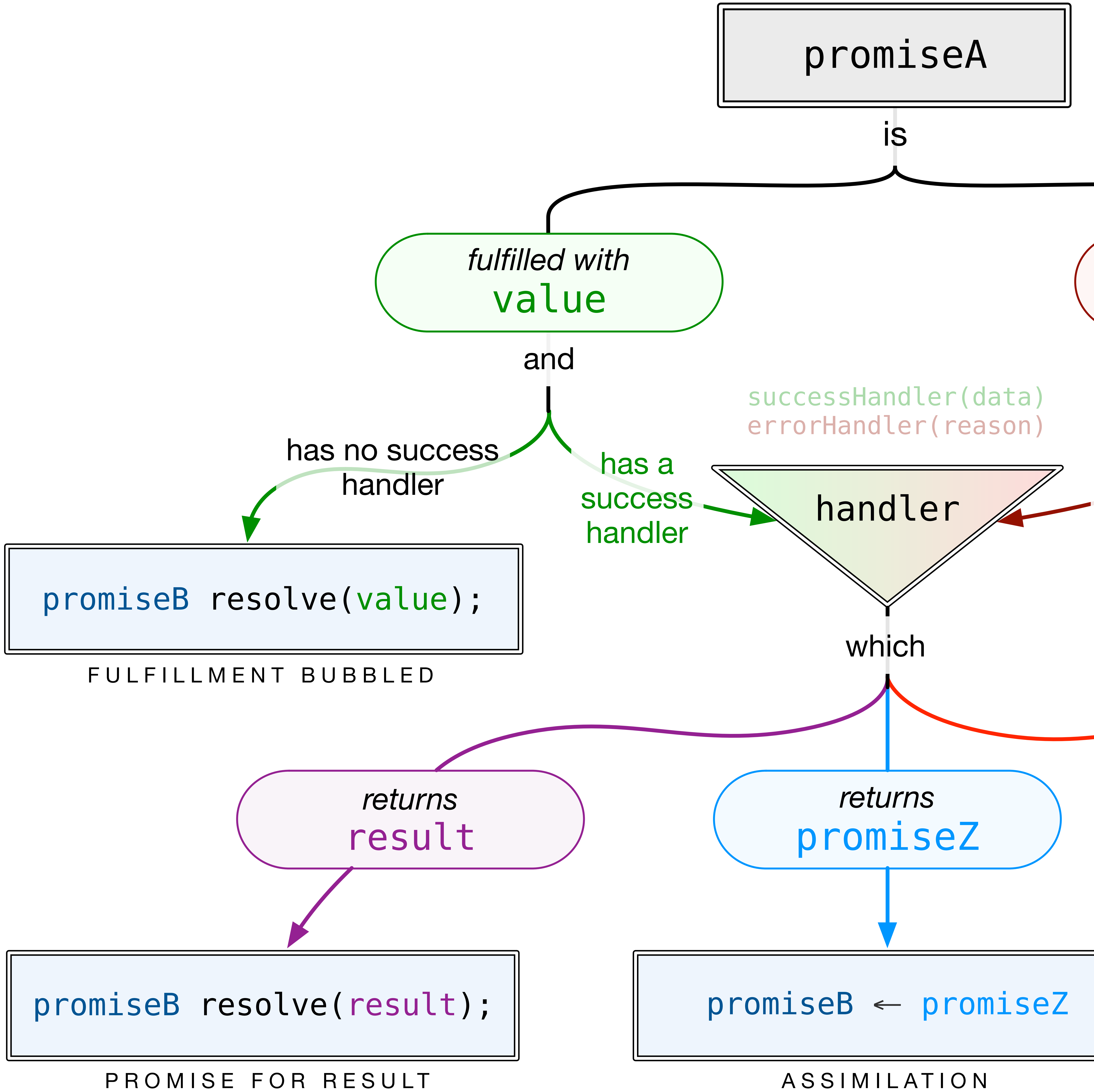
promiseB reject(err);

ERROR CAUGHT

```
// promiseA fulfills with 'Hello.'

promiseA
  .then() // -> p1
  .then() // -> p2
  .then() // -> p3
  .then() // -> p4
  .then() // -> p5
  .then(blue);
```

Fulfillment bubbled down to first available success handler:

Console says "Hello."



promiseA

is

fulfilled with value

and

has no success handler

has a success handler

successHandler(data)

errorHandler(reason)

handler

promiseB resolve(value);

FULFILLMENT BUBBLED

which

returns result

returns promiseZ

promiseB resolve(result);

PROMISE FOR RESULT

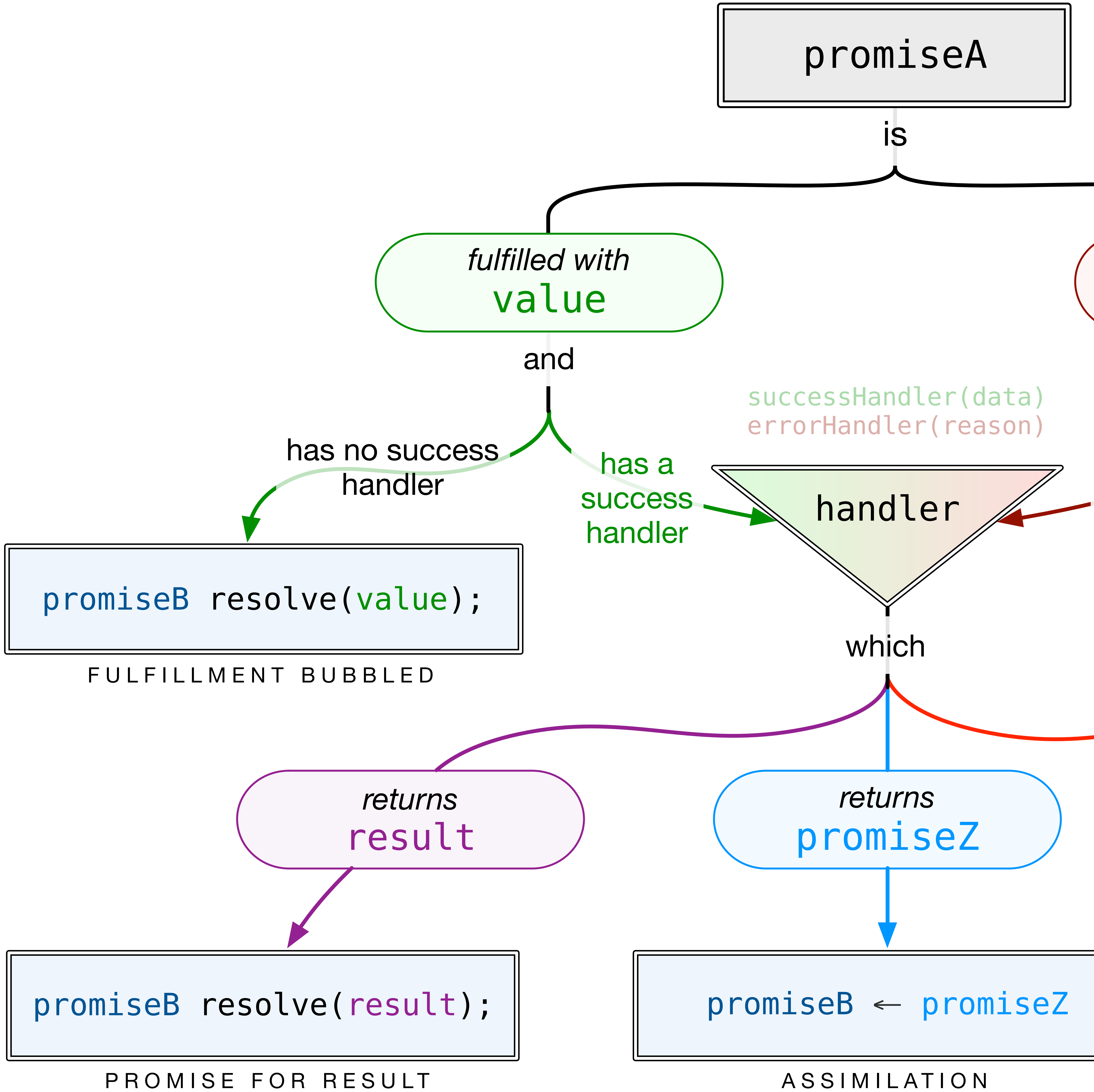
promiseB ← promiseZ

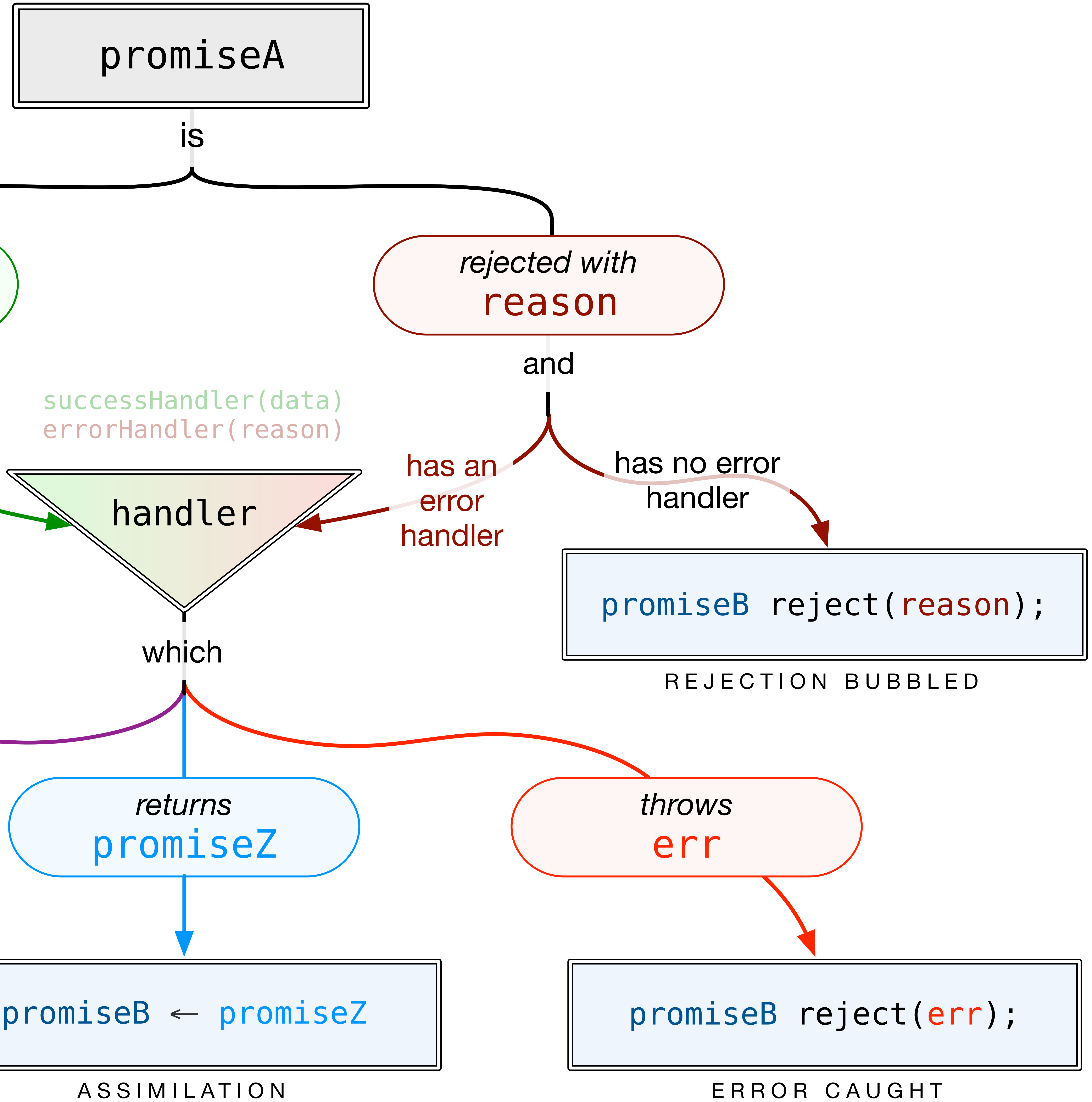
ASSIMILATION

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

promiseA
  .then(null, warnUser) // -> p1
  .then()               // -> p2
  .then()               // -> p3
  .then(null, null)     // -> p4
  .then()               // -> p5
  .then(blue);
```

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

Console log reads "Hello."

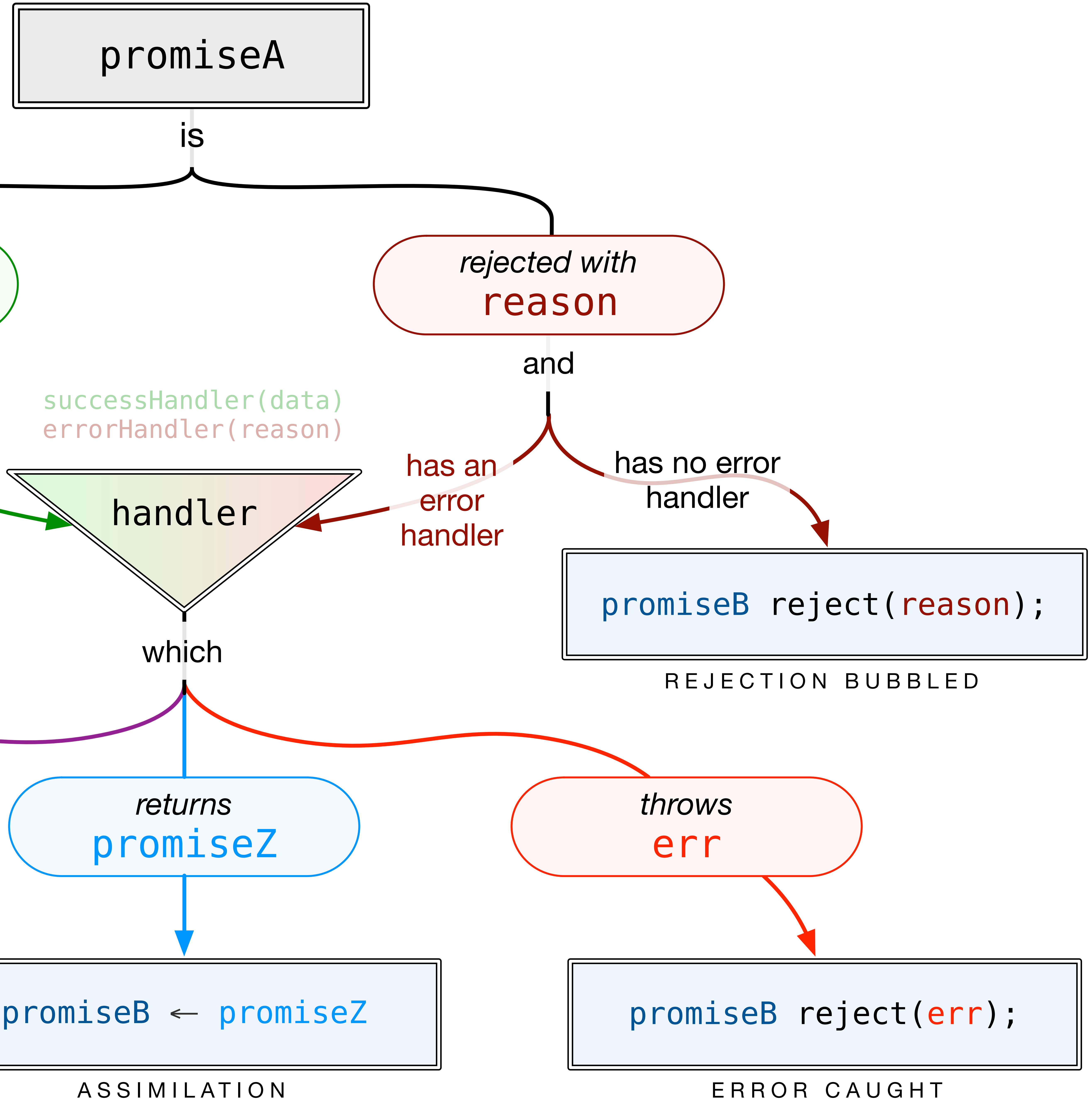




```
function logYell (input) {
  console.log(input+'!');
}

// promiseA rejected with ‘Sorry'

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

Rejection bubbles down to the first available error handler.

Console log is “Sorry”.



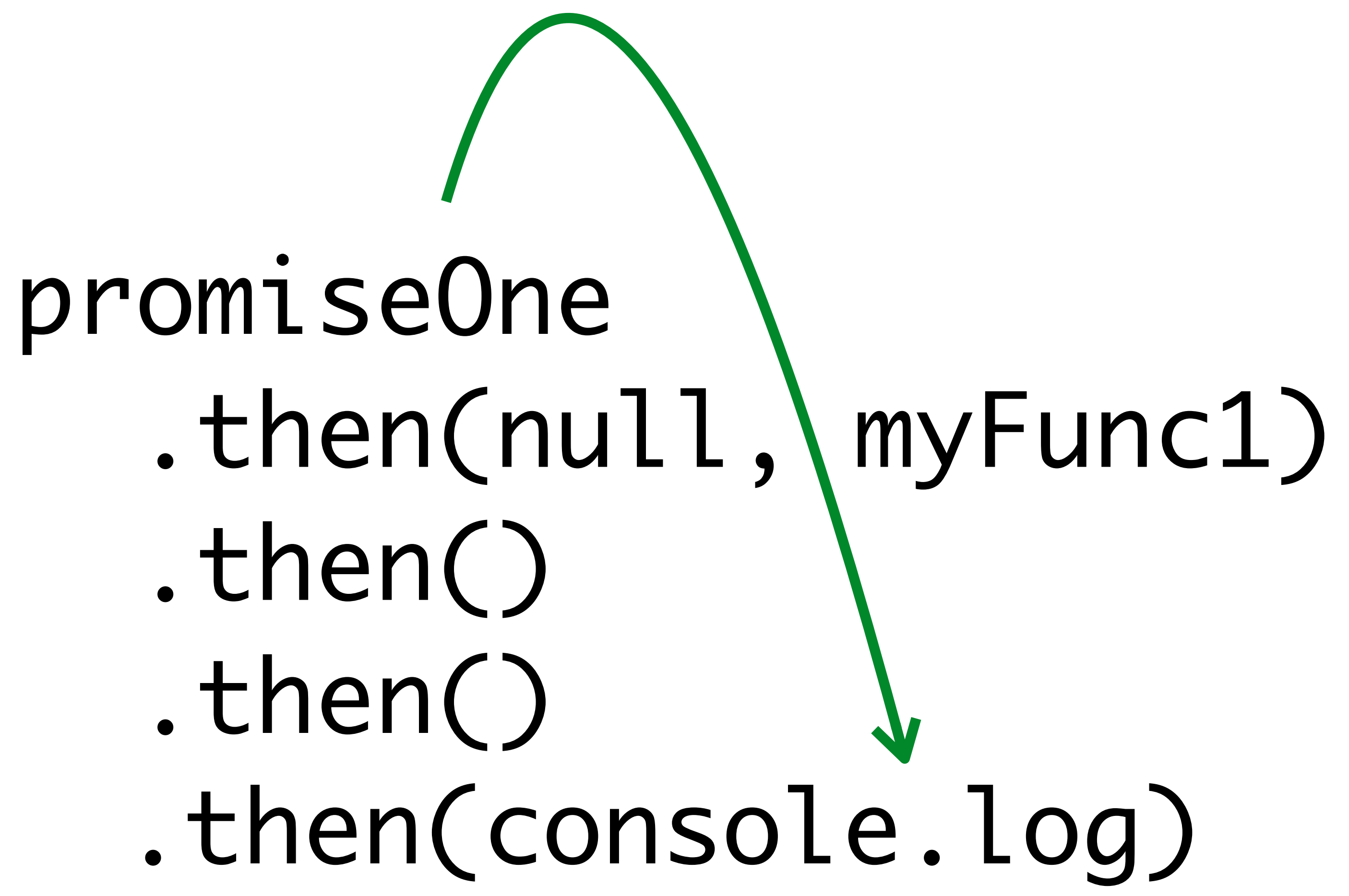
```
function logYell (input) {
  console.log(input+'!');
}

// promiseA rejected with ‘Sorry'

promiseA
  .then(boundLog)     // -> p1
  .then()             // -> p2
  .then(null, null)   // -> p3
  .then(null, logYell);
```

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

Console log is "Sorry!"

Review: Success & Error Bubbling

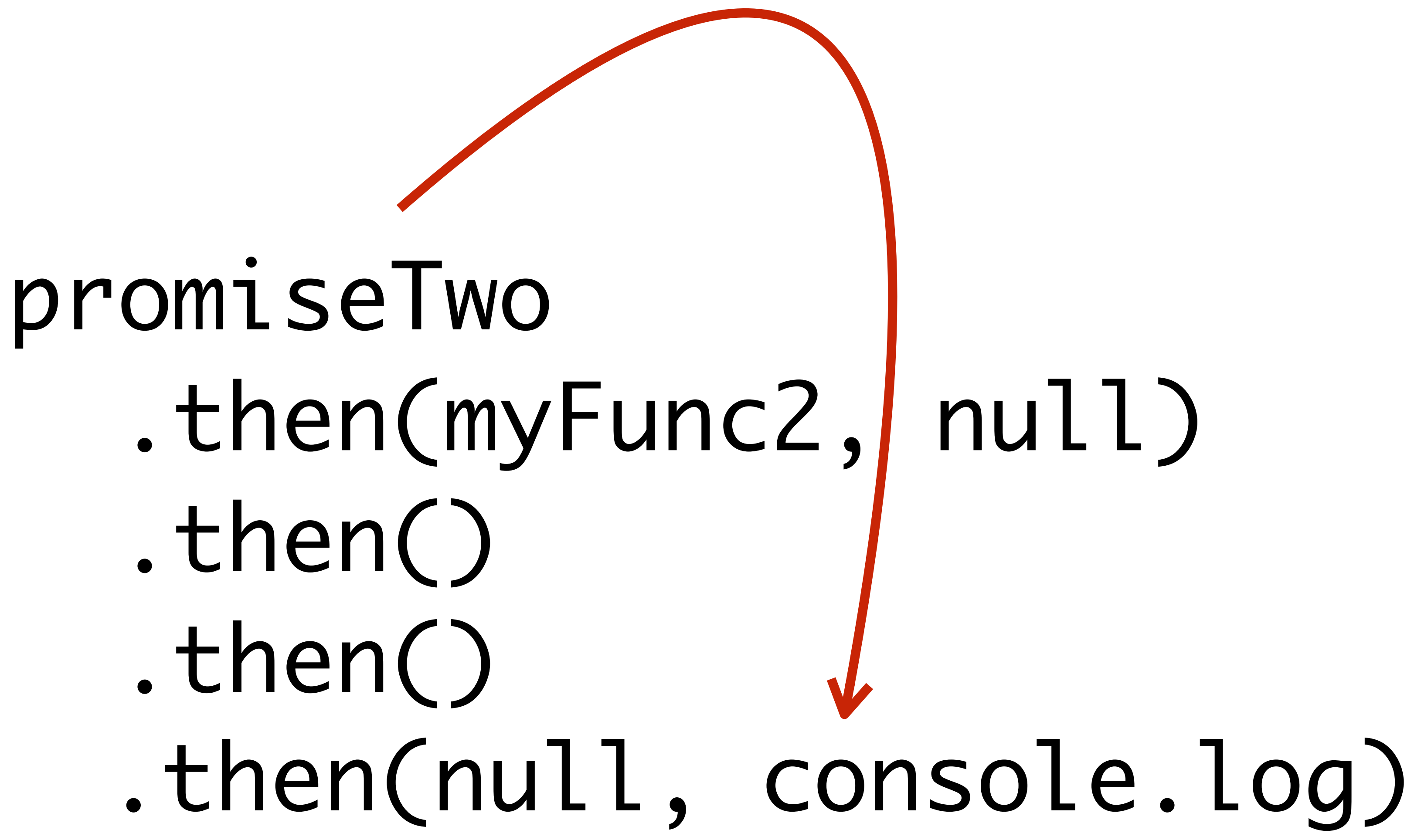
```
// promiseOne is fulfilled
with ‘hello’
```



```
promiseOne
  .then(null, myFunc1)
  .then()
  .then()
  .then(console.log)
```

```
// result: console shows
‘hello’
// fulfilled value bubbled
to success handler
```

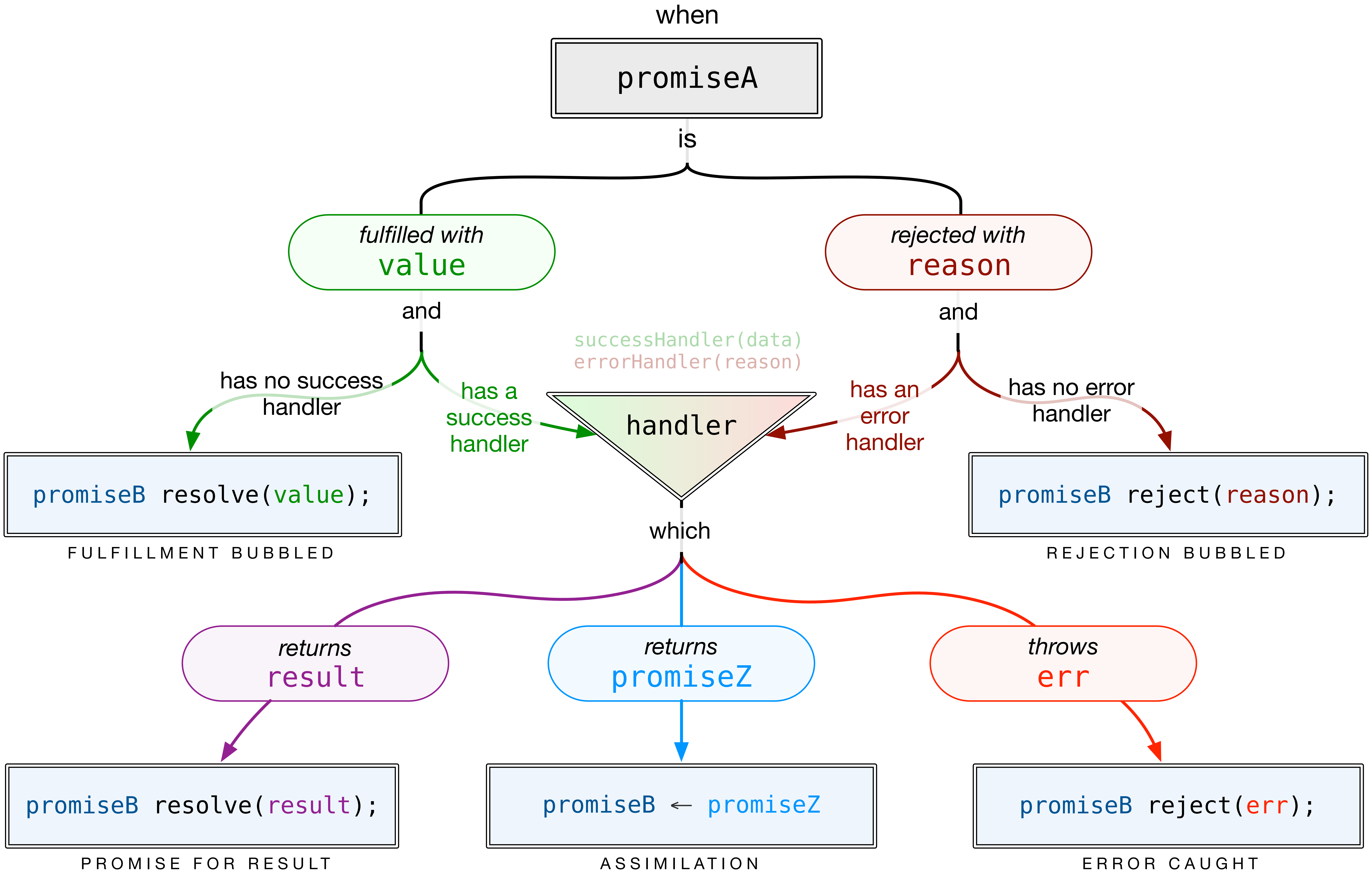
```
// promiseTwo is rejected
with ‘bad request’
```



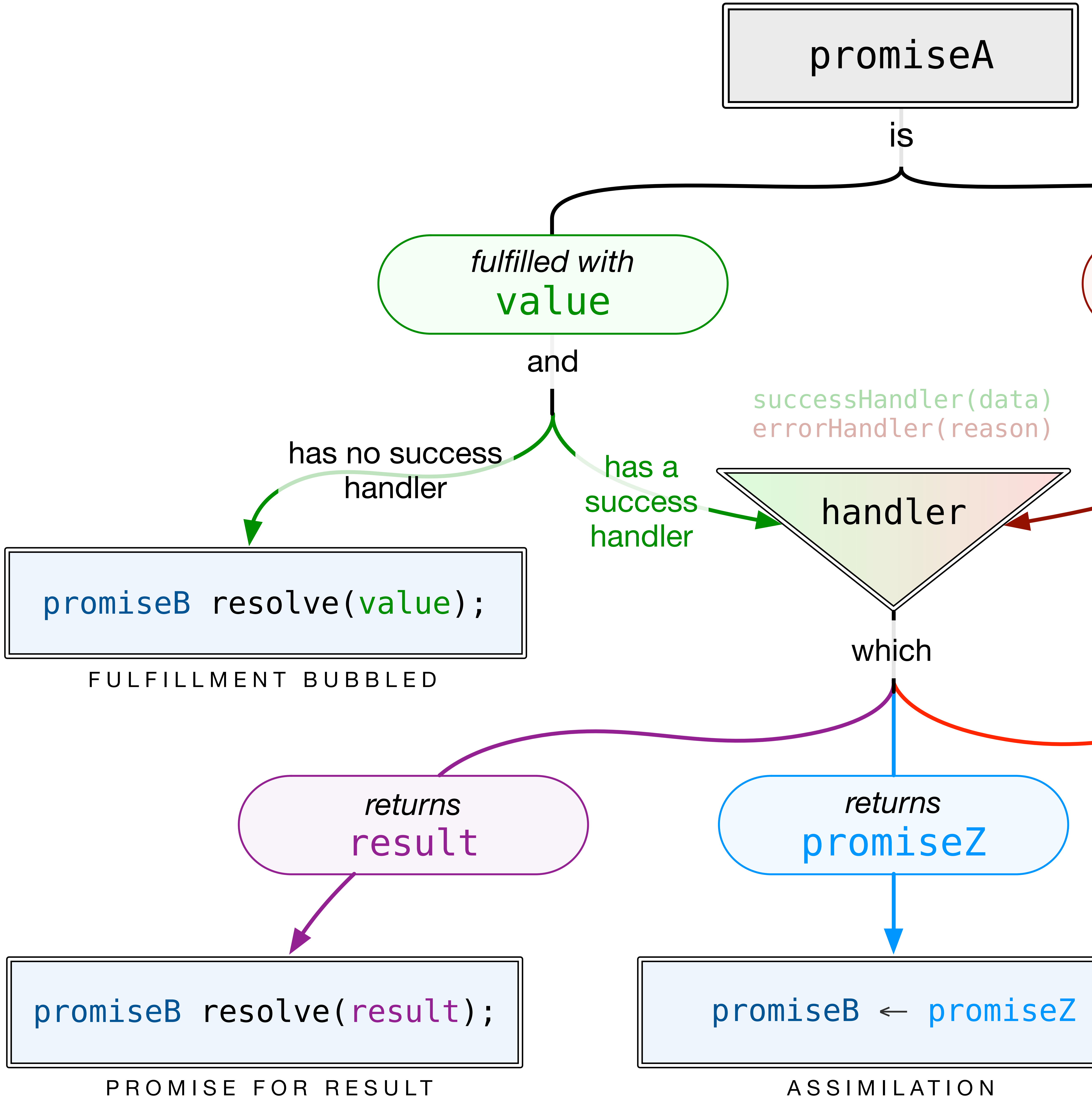
```
promiseTwo
  .then(myFunc2, null)
  .then()
  .then()
  .then(null, console.log)
```

```
// result: console shows
‘bad request’
// rejection bubbled to
error handler
```

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



```
promiseA
  .then(function(valueOne){
    // do some code to make valueTwo
    return valueTwo
  })
  .then(function(valueTwo){
    console.log(valueTwo)
  })
```



promiseA

is

fulfilled with
value

and

has no success handler

has a success handler

successHandler(data)
errorHandler(reason)

handler

which

promiseB resolve(value);

FULFILLMENT BUBBLED

returns
result

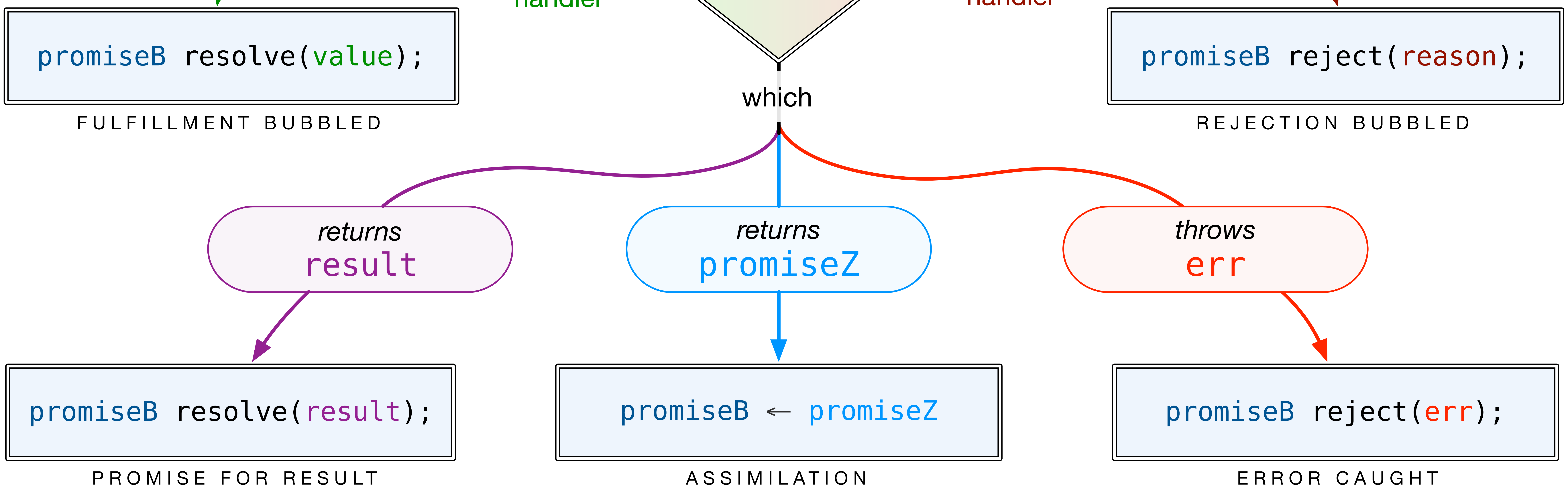
returns
promiseZ

promiseB resolve(result);

PROMISE FOR RESULT

promiseB ← promiseZ

ASSIMILATION



```
promiseA
  .then(function(valueOne){
    // do some code to make a
    // totally new promise,
    return someTotallyNewPromise
  })
  .then(function(someValue){
    console.log(someValue)
  })
```

Review: Returning from Handler

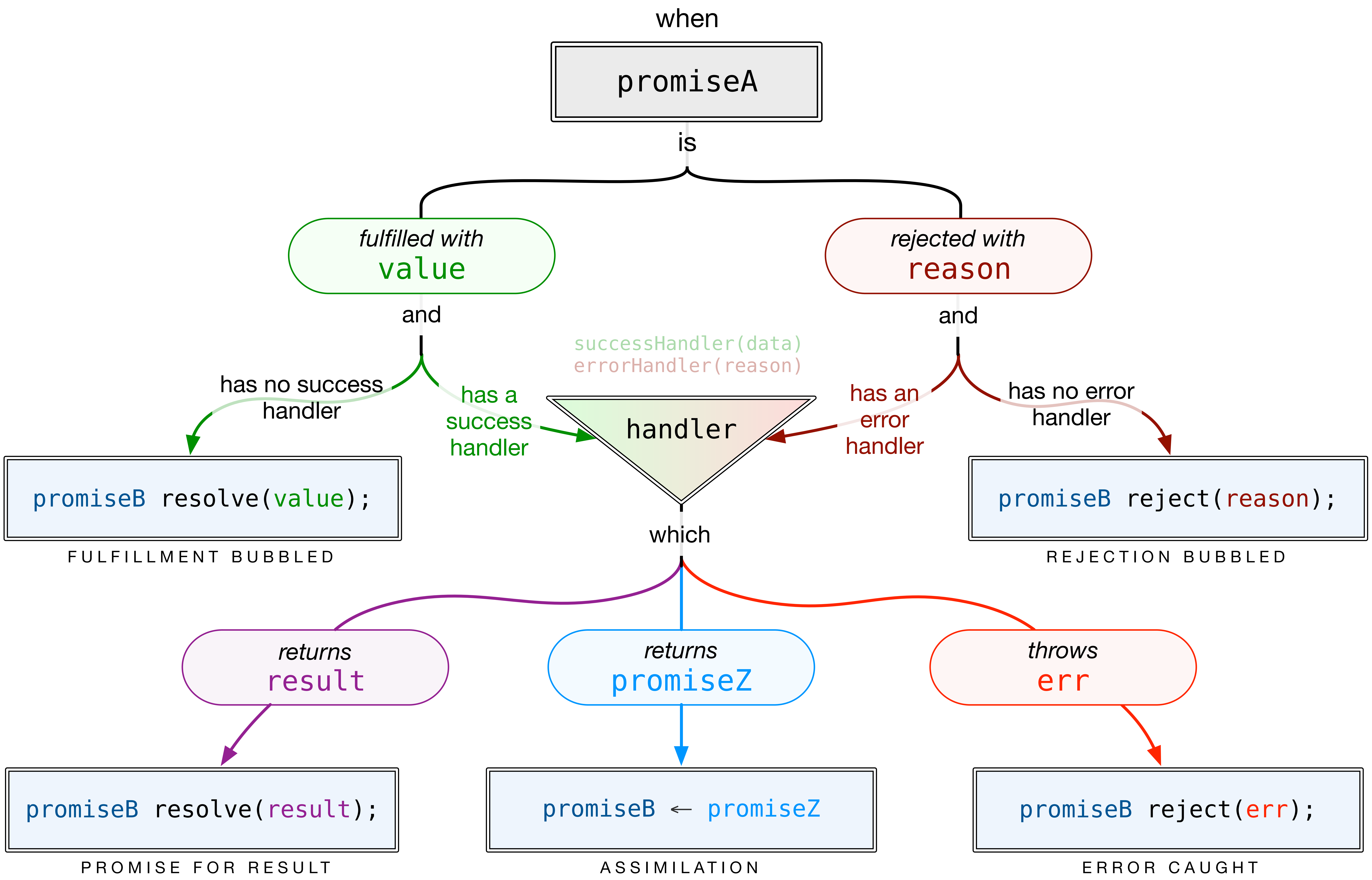
```
// with a value

promiseOne
  .then(function(valueOne){
    // do some code to make valueTwo
    return valueTwo
  })
  .then(function(valueTwo){
    console.log(valueTwo)
  })
```

```
// with a promise

promiseTwo
  .then(function(valueOne){
    // do some code to make a
    // totally new promise,
    return someTotallyNewPromise
  })
  .then(function(someValue){
    console.log(someValue)
  })
```

The Magic of Promises

- **Magical Property #1: It doesn’t really matter whether .then() is called before or after the promise is resolved. Everything just works properly!**
- **Magical Property #2: .then() returns a new (different) Promise.**
  - whatever the previous promise returned ends up as the resolved value in the new promise.
  - If the previous promise returned a promise… then the **resolved value** of the returned promise ends up in the new one. Woah.

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



when

promiseA

is

fulfilled with
value

and

has no success handler

promiseB resolve(value);

FULFILLMENT BUBBLED

has a success handler

rejected with
reason

and

has an error handler

has no error handler

promiseB reject(reason);

REJECTION BUBBLED

successHandler(data)
errorHandler(reason)

handler

which

returns
result

promiseB resolve(result);

PROMISE FOR RESULT

returns
promiseZ

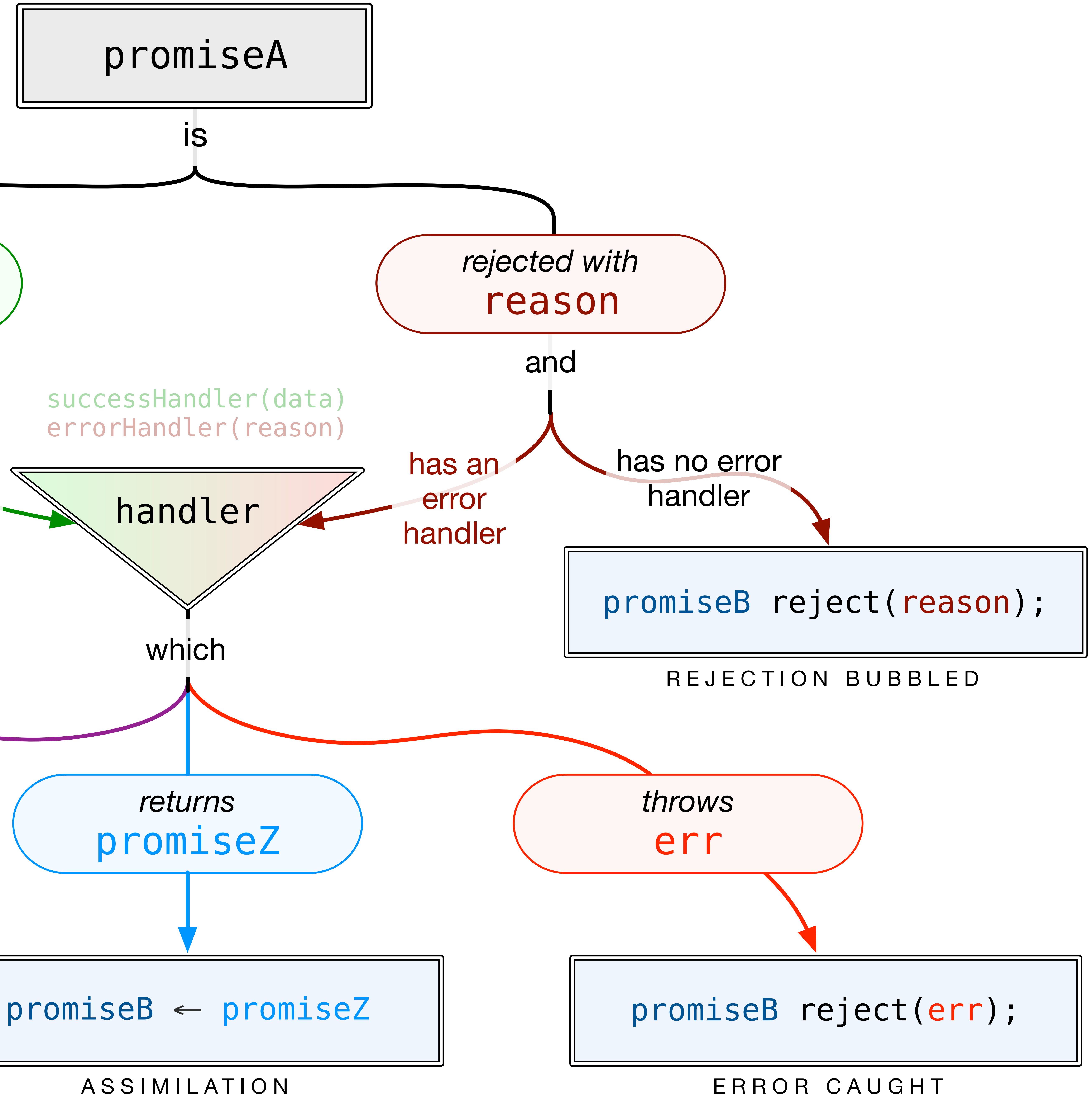
promiseB ← promiseZ

ASSIMILATION

throws
err

promiseB reject(err);

ERROR CAUGHT



promiseA

is

rejected with
reason

and

successHandler(data)
errorHandler(reason)

has an error handler

has no error handler

handler

promiseB reject(reason);

REJECTION BUBBLED

which

returns
promiseZ

throws
err

promiseB ← promiseZ

ASSIMILATION

promiseB reject(err);

ERROR CAUGHT

```
promiseA
  .then(function(valueOne){
    // OH NO! THROW AN ERROR, 404
    return valueOne
  })
  .then(null, function(err){
    console.error(err)
  })
```

that was .then

```
// array of API calls to make
const 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:
const thingsPromise = Promise.all( apiCallPromises );
// use it:
thingsPromise.then( function got (results) {
  results.forEach( function print (result) {
    console.log(result);
  });
});
```

Node.js promises: native & Bluebird

```
Promise // built-in
```

```
npm install bluebird --save

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

(some) Sequelize Promises

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

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

const savedUserPromise = gandalf.save();

const userIsDestroyedPromise = gandalf.destroy();

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

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

External Resources for Further Reading

- 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)
- DailyJS: Javascript Promises in Wicked Detail (build an ES6-style implementation)
- MDN: ES6 Promises (upcoming native functions)
- Promise Nuggets (use patterns)
- Promise Anti-Patterns

I Will Be Able To…

- Explain the behaviour of promises under multiple conditions (e.g. chaining on same promise, chaining without a handler, returning a new promise)
- Build my own promise library