Mechanics of Promises

Understanding JavaScript Promise Generation & Behavior

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); },
  next
);
```

Why so awesome?!

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!

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

// promise is portable - can move it around
// (in this file or any file!)
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(days => res.render('calendar', {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));
```

So, what is a promise?

Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems

MIT: 1988

Barbara Liskov

Liuba Shrira

"A promise represents the eventual result of an asynchronous operation."

— THE PROMISES/A+ SPEC


magic!
(no)
iHDWALLPAPERS.COM

Promises are Objects

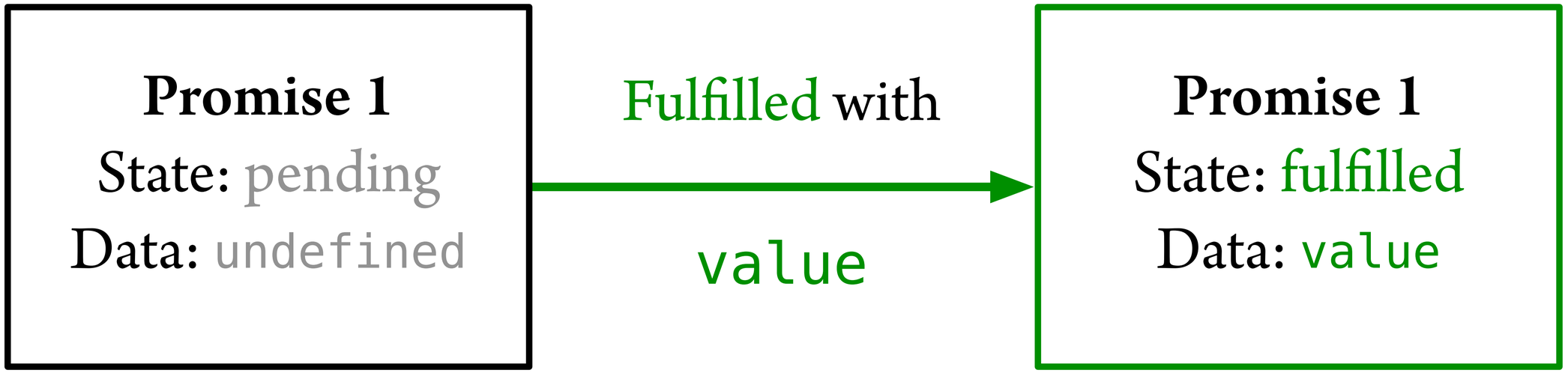
state (pending, fulfilled, or rejected)

information (value or a reason)

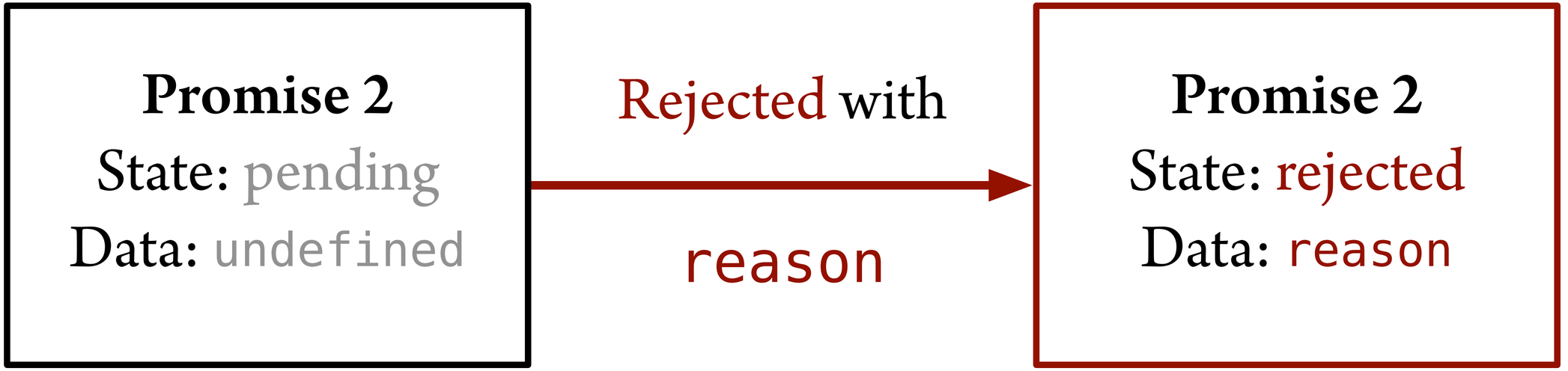
} (hidden if possible)

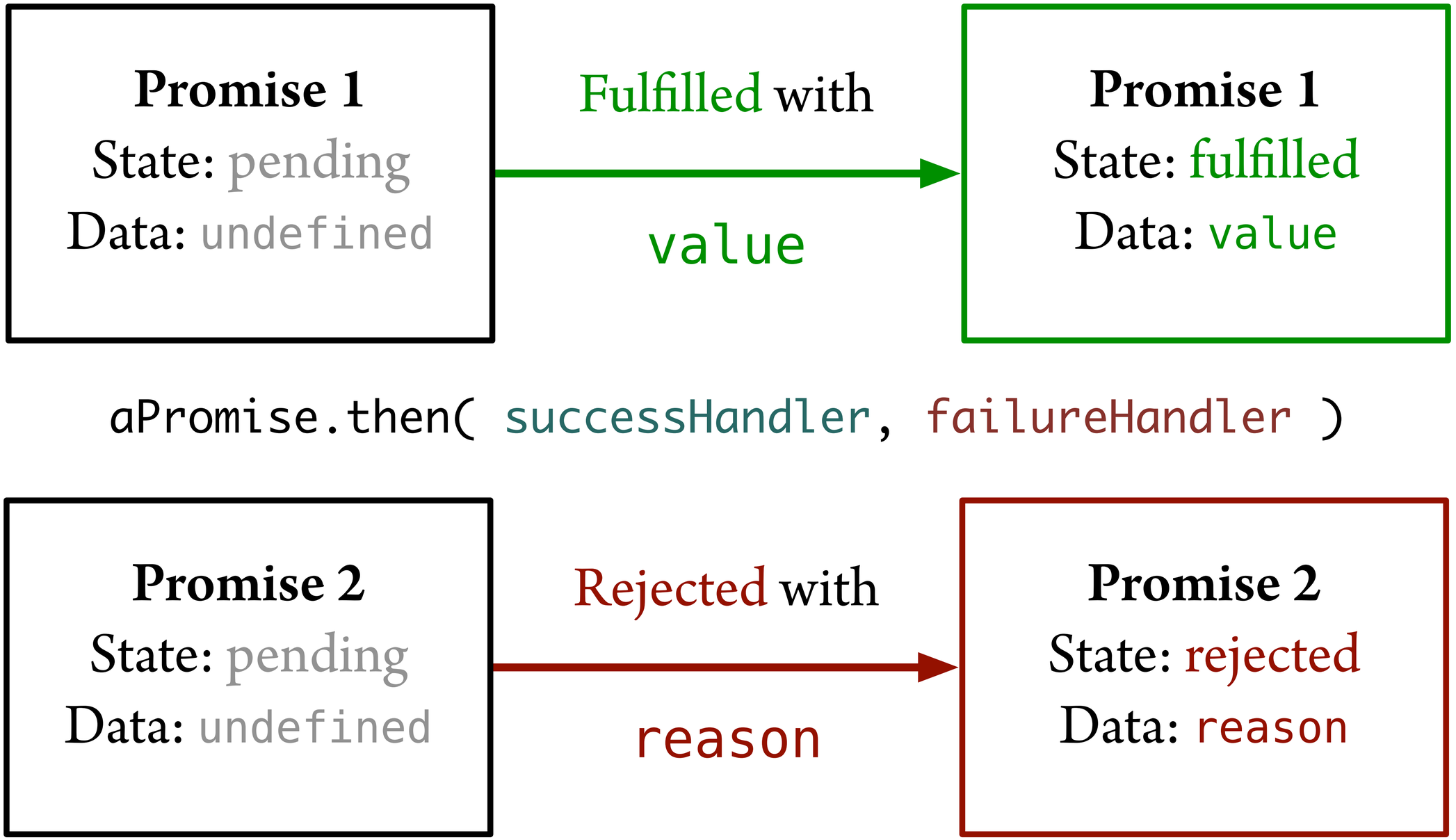
```
.then()
.catch()
```

} (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

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

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

- 1. **Promise settles**
  2. *add handler*
  3. handler is called once

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

- Can attach handlers at multiple times (different modules even)

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"

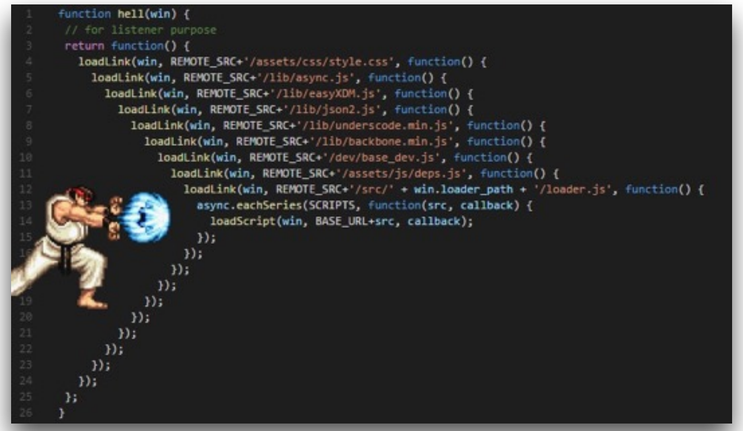
Callback Hell

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

const 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

const 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);
  });
```

Black Holes

you literally cannot return from a typical async callback

```
// This will not work!

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

```
// This also will not work

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

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

```
// Fantasy solution

const 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

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 P/A+

So where do real promises come from?

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

Making New Promises: How?

Minimal Examples of Promise Construction

```
const promiseFor123 = new Promise(resolve => {
  resolve( 123 );
});

const promiseFor000 = new Promise((resolve, reject) => {
  reject( 000 );
});
```

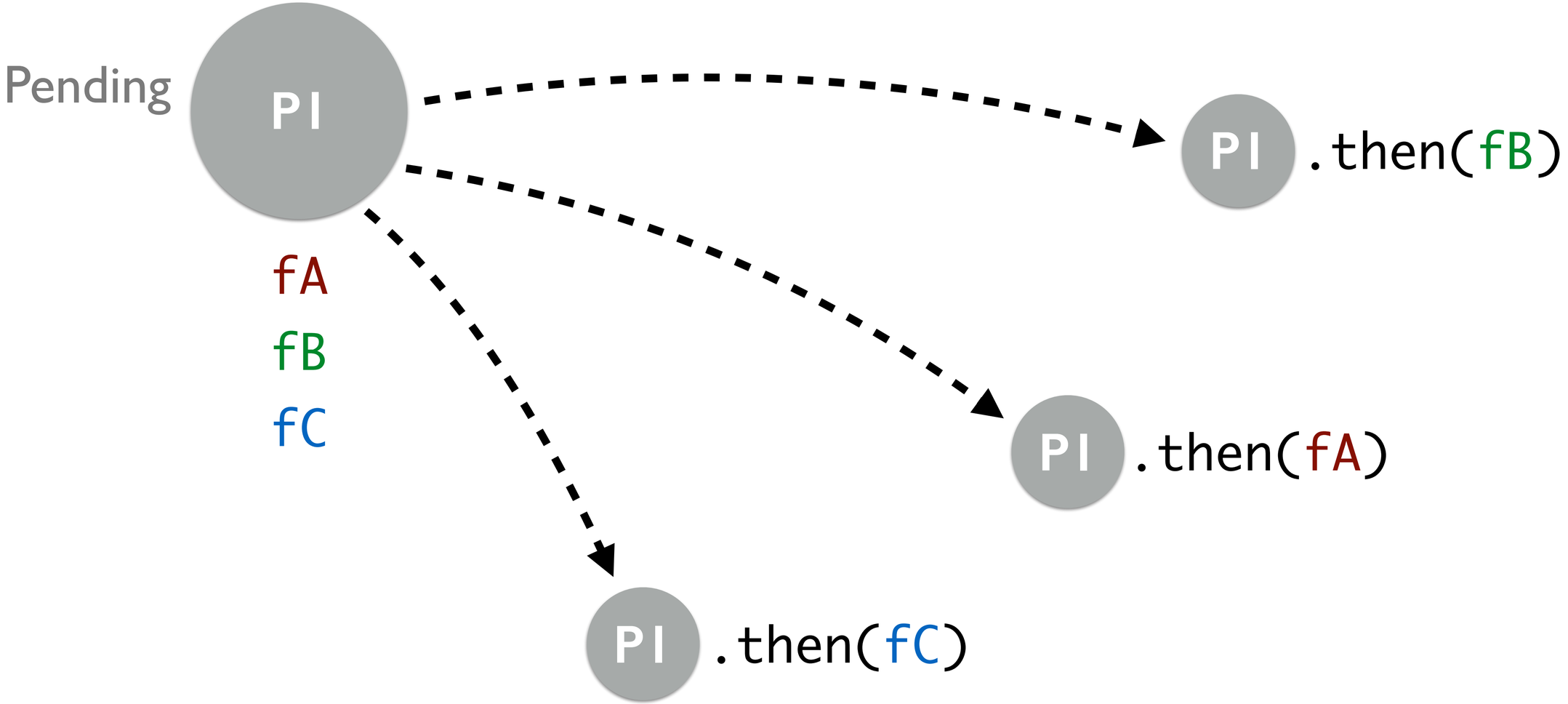
Promisification in Node.js

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

readFilePromise(‘path’).then( someSuccessHandler, someErrorHandler );
```

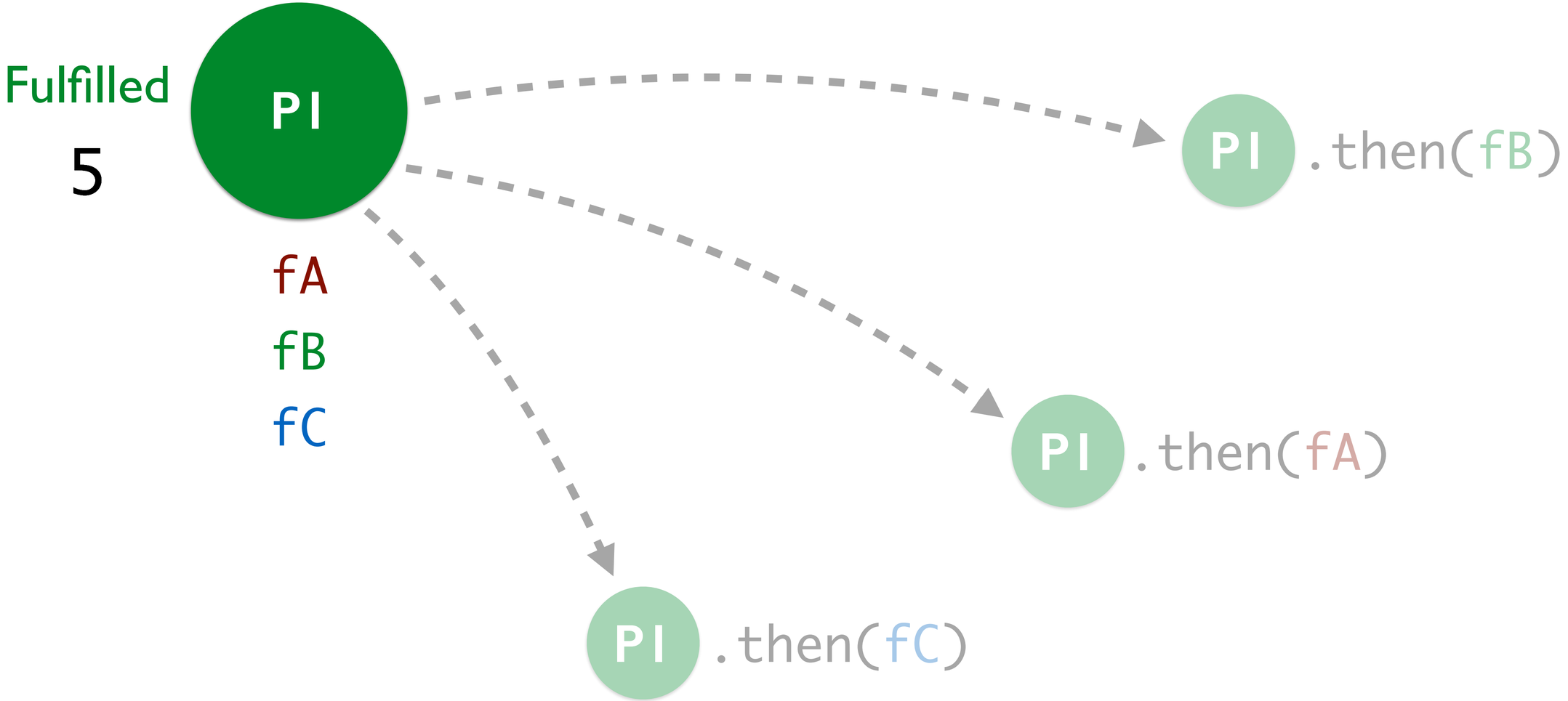
```
Bluebird.promisifyAll( fs );
fs.readFileAsync(‘path').then(function (text) {
  // use the text
});
```

(Break)

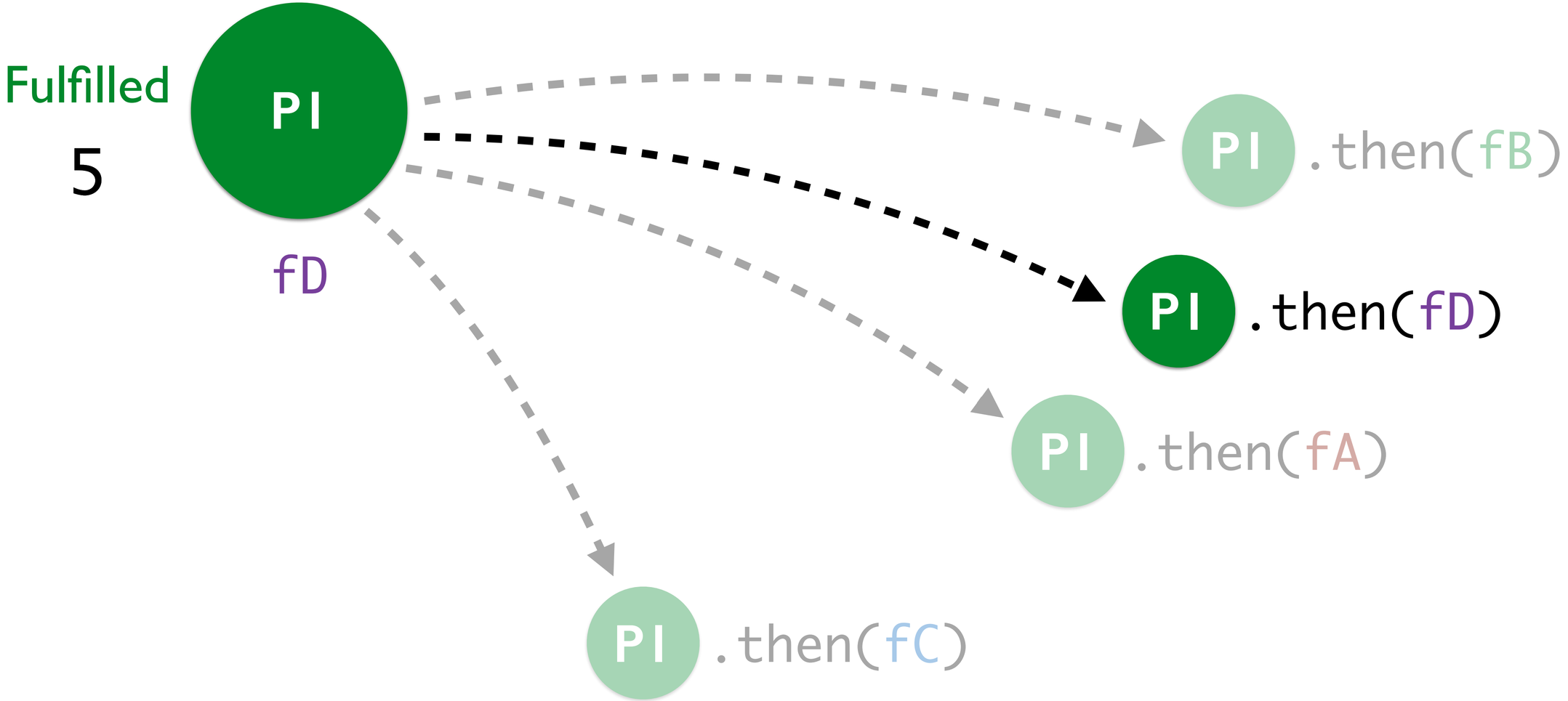
.then on same promise (not chaining!)



.then on same promise (not chaining!)



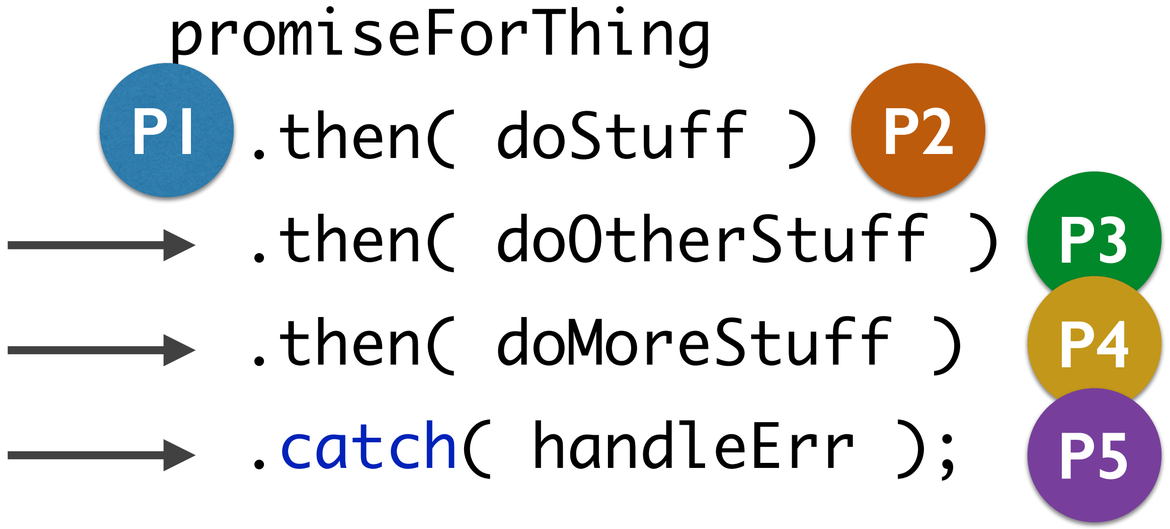
.then on same promise (not chaining!)



the magic: .then returns a *new* promise

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

This is why we can chain .then



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

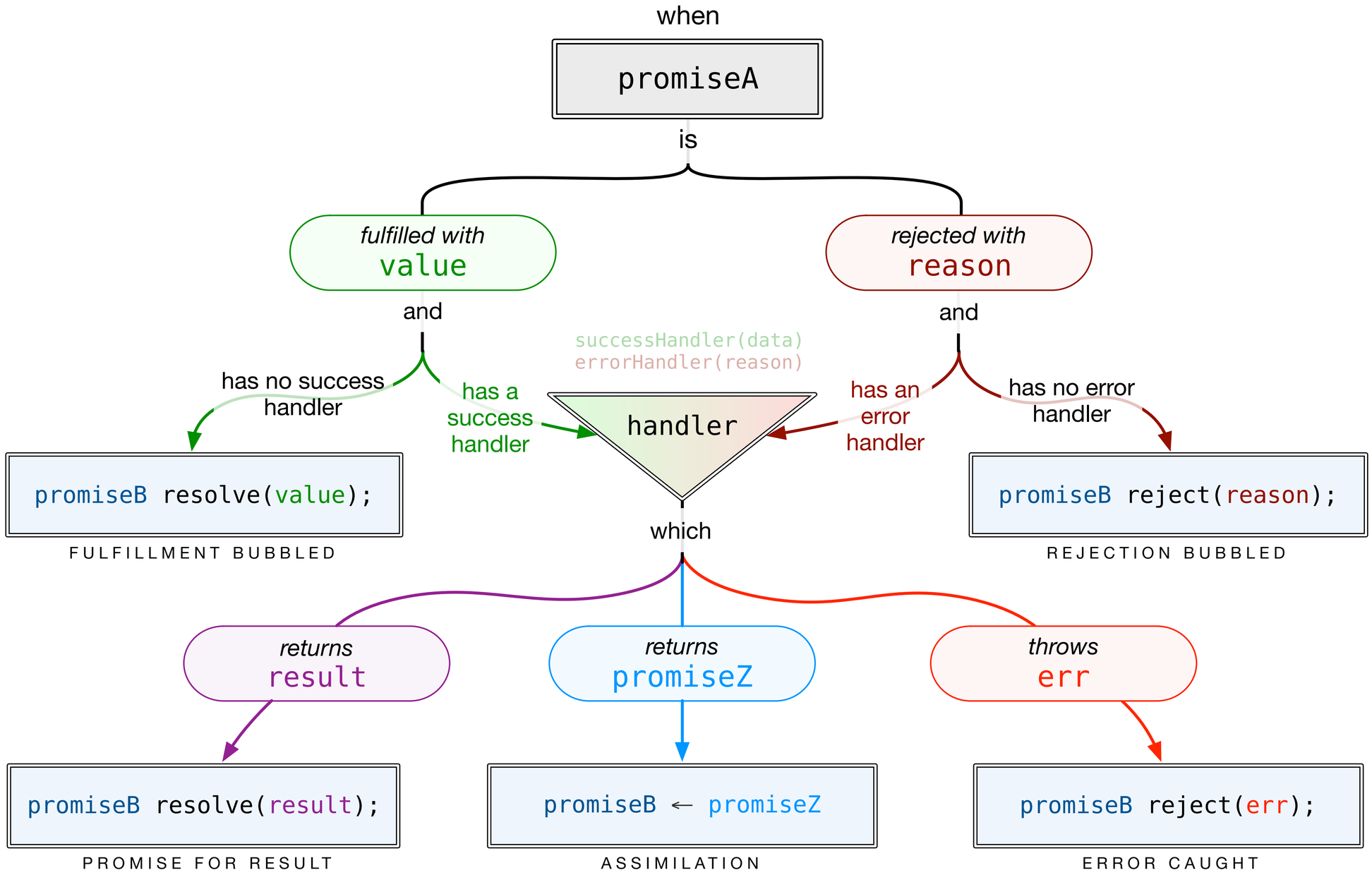
And why we can `return` in a handler

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

What is promiseB a promise for?

Brace yourselves...

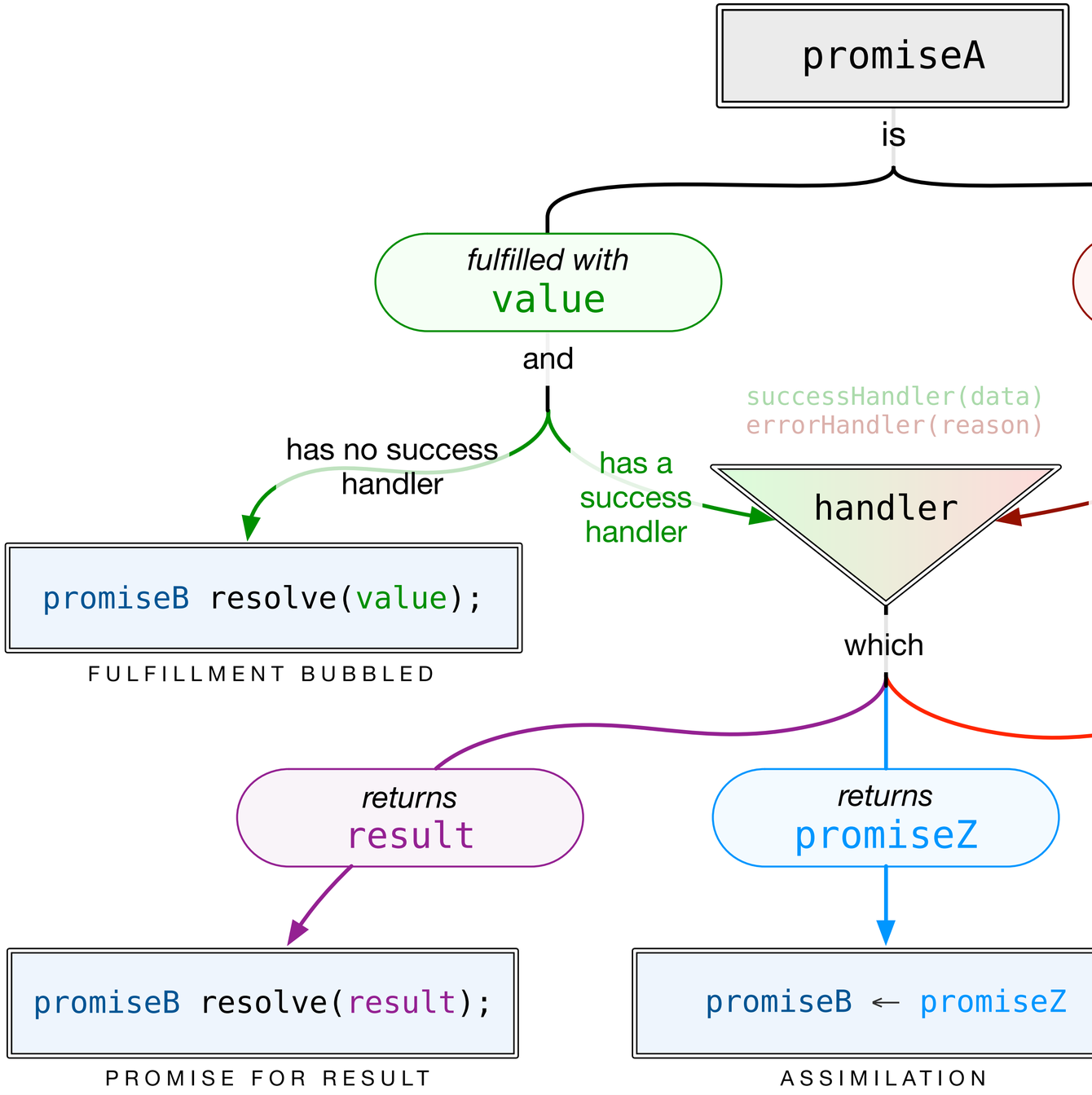
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



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

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

Fulfillment bubbled down to first available success handler:

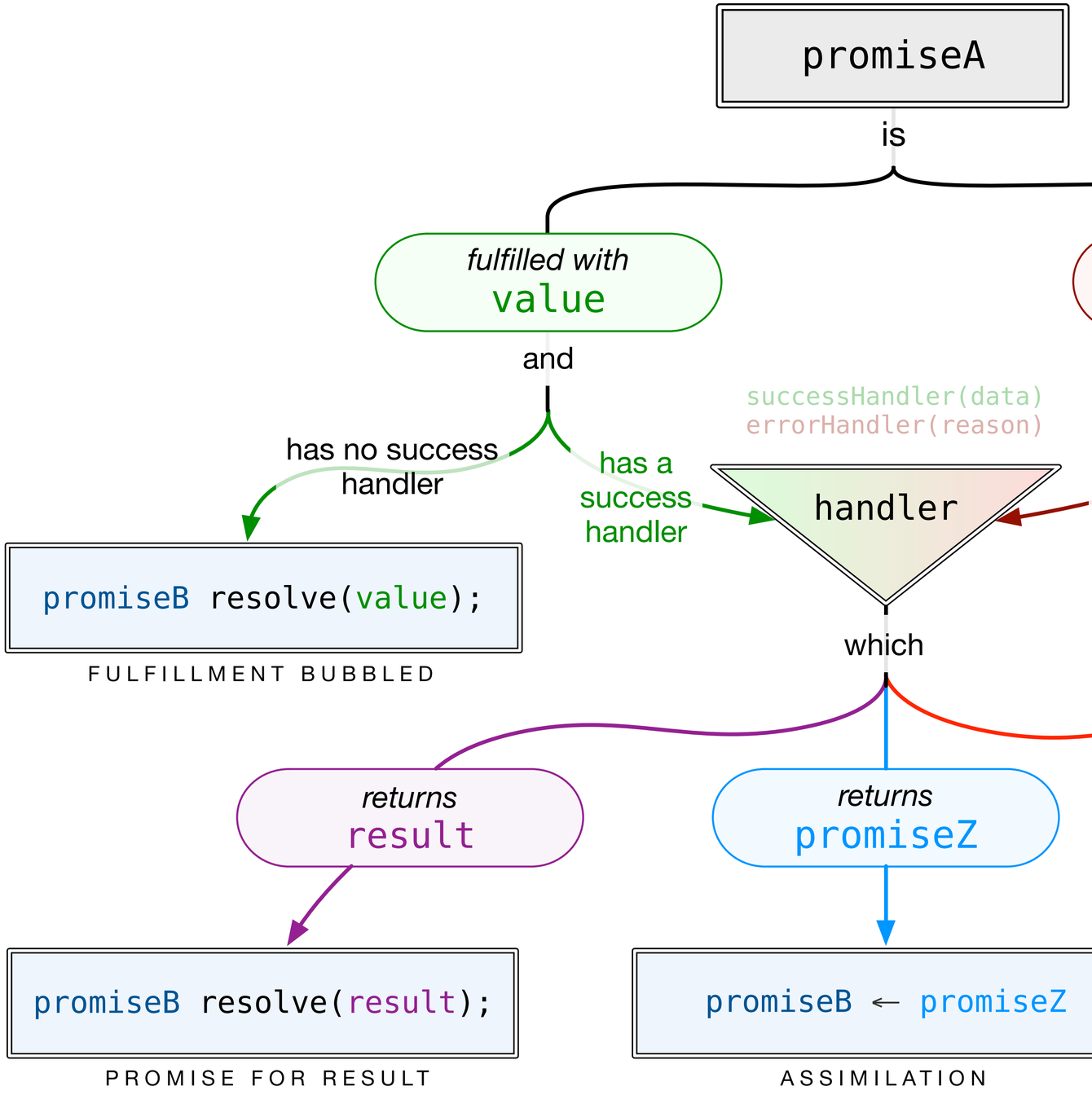
Console log reads “Hello.”

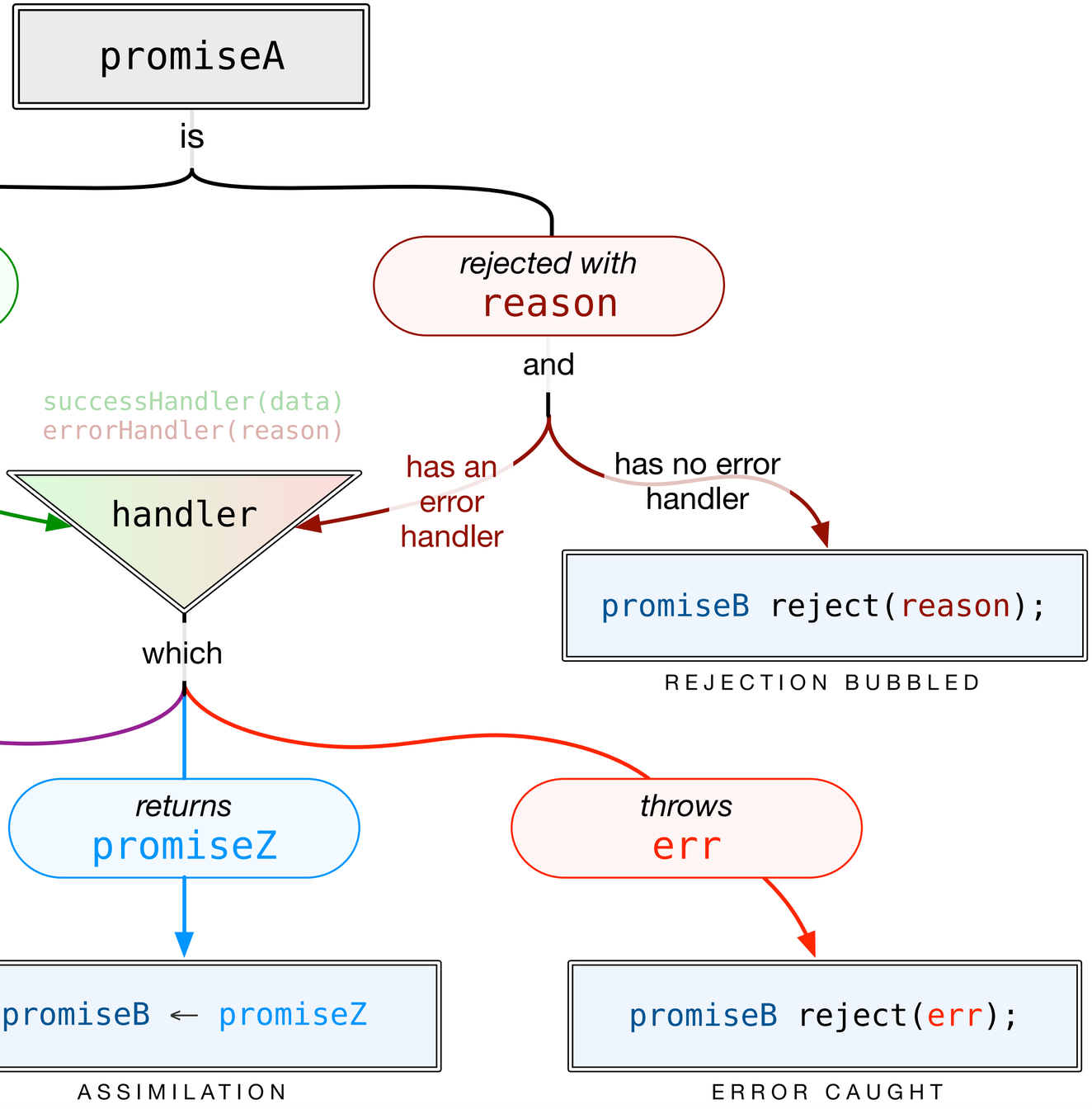


```
// promise0 fulfills with 'Hello.'

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

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.”

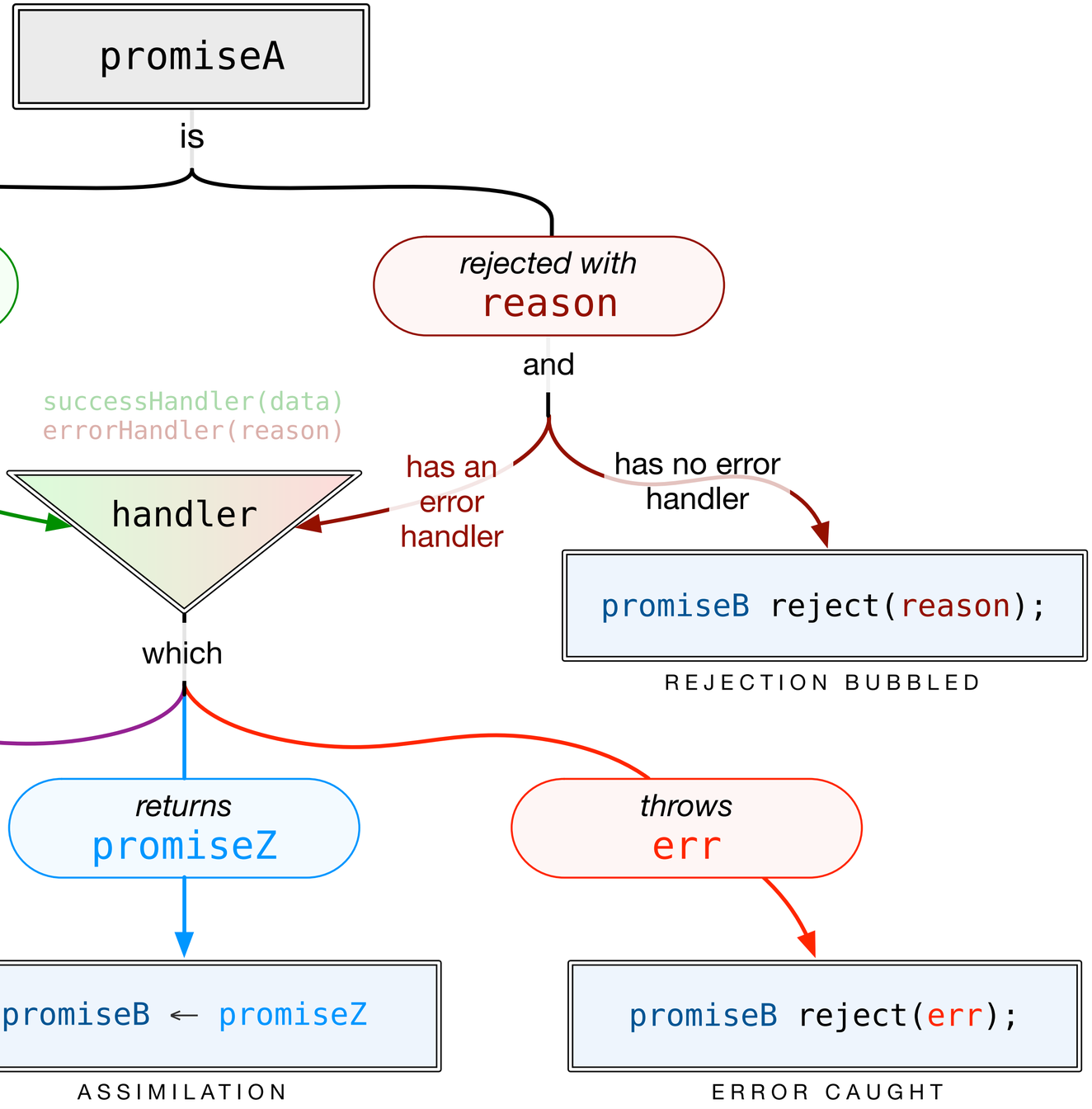





```
function logYell (input) {
  console.log(input+'!');
}

// promise0 rejected with ‘Sorry'

promise0
  .then() // -> p1
  .then() // -> p2 and so on
  .then()
  .then(null, boundLog);
```

Rejection bubbles down to the first available error handler.

Console log is “Sorry”.



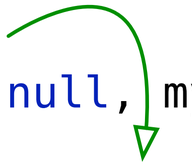

```
function logYell (input) {
  console.log(input+'!');
}

// promise0 rejected with 'Sorry'

promise0
  .then(boundLog) // -> p1
  .then() // -> p2 and so on
  .then(null, null)
  .then(null, logYell);
```

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

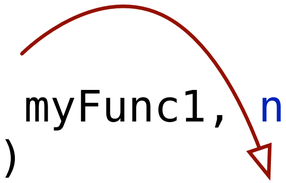
Console log is "Sorry!"

Review: Success & Error Bubbling

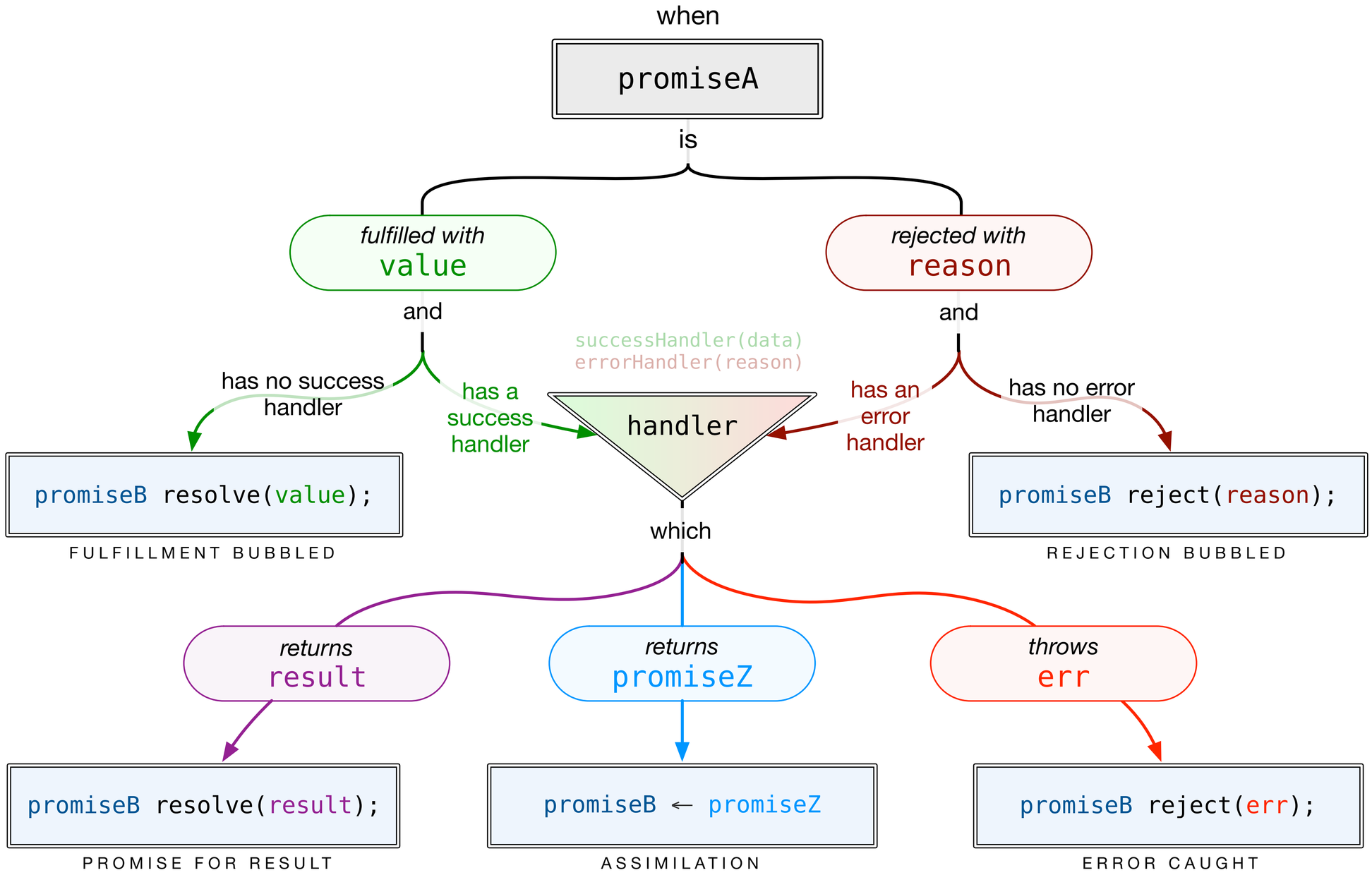
```
// promiseA is fulfilled with 'hello'

promiseA
  .then( null, myFunc1, myFunc2 )
  .then()
  .then( console.log );

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



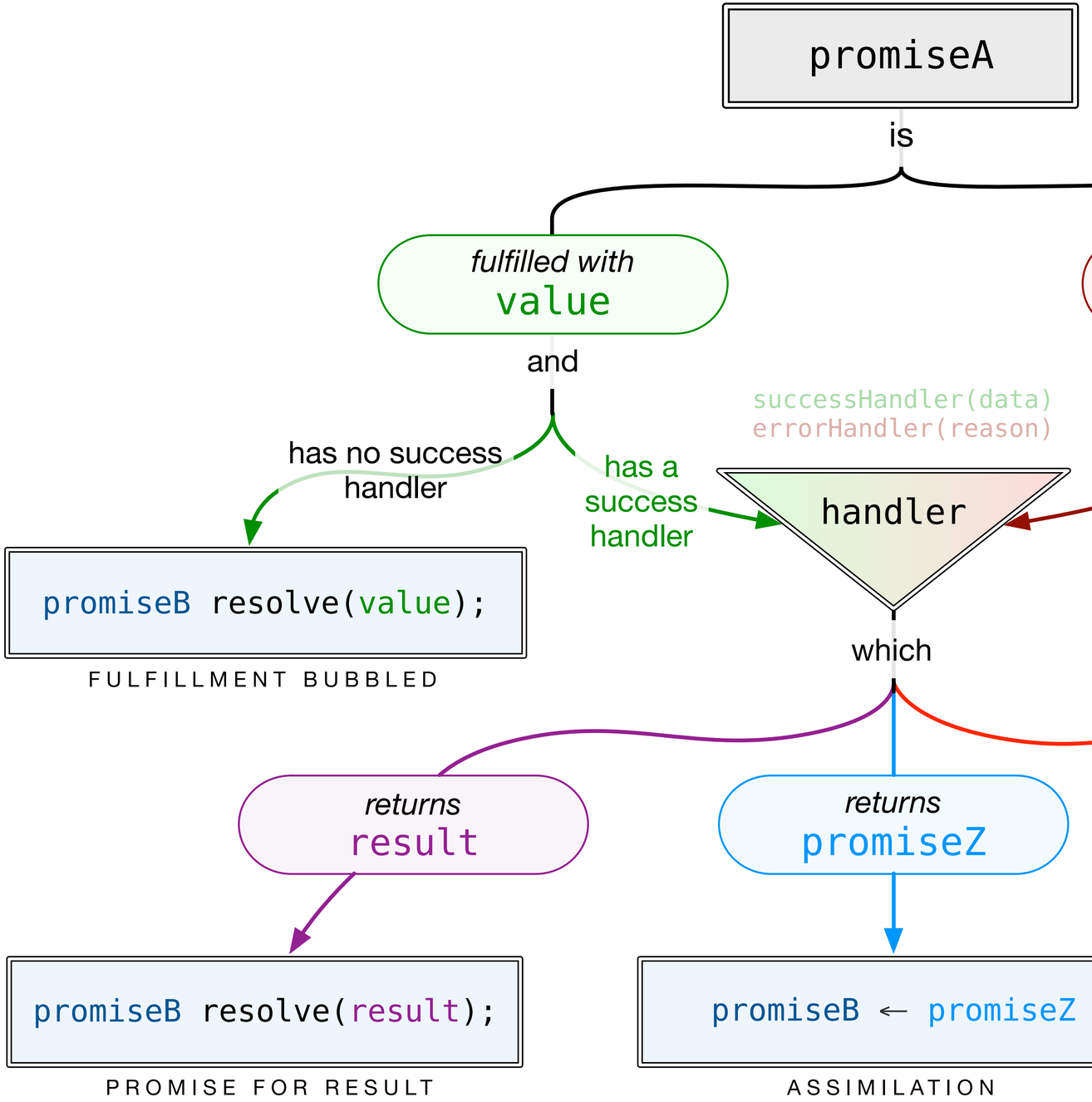
```
// promiseA is rejected with 'bad request'

promiseA
  .then( myFunc1, null, myFunc2 )
  .then()
  .then( null, console.log );

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



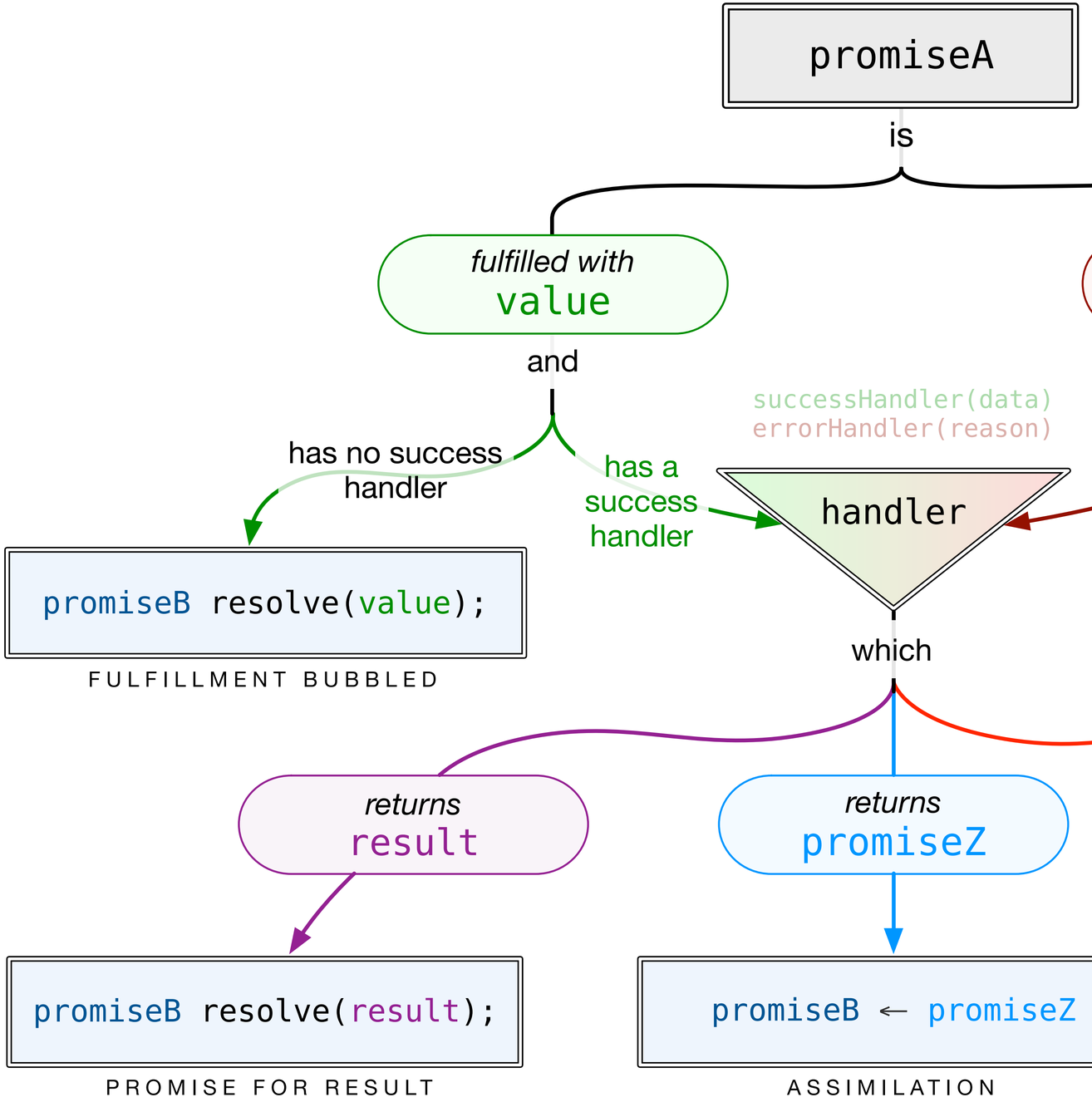
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



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

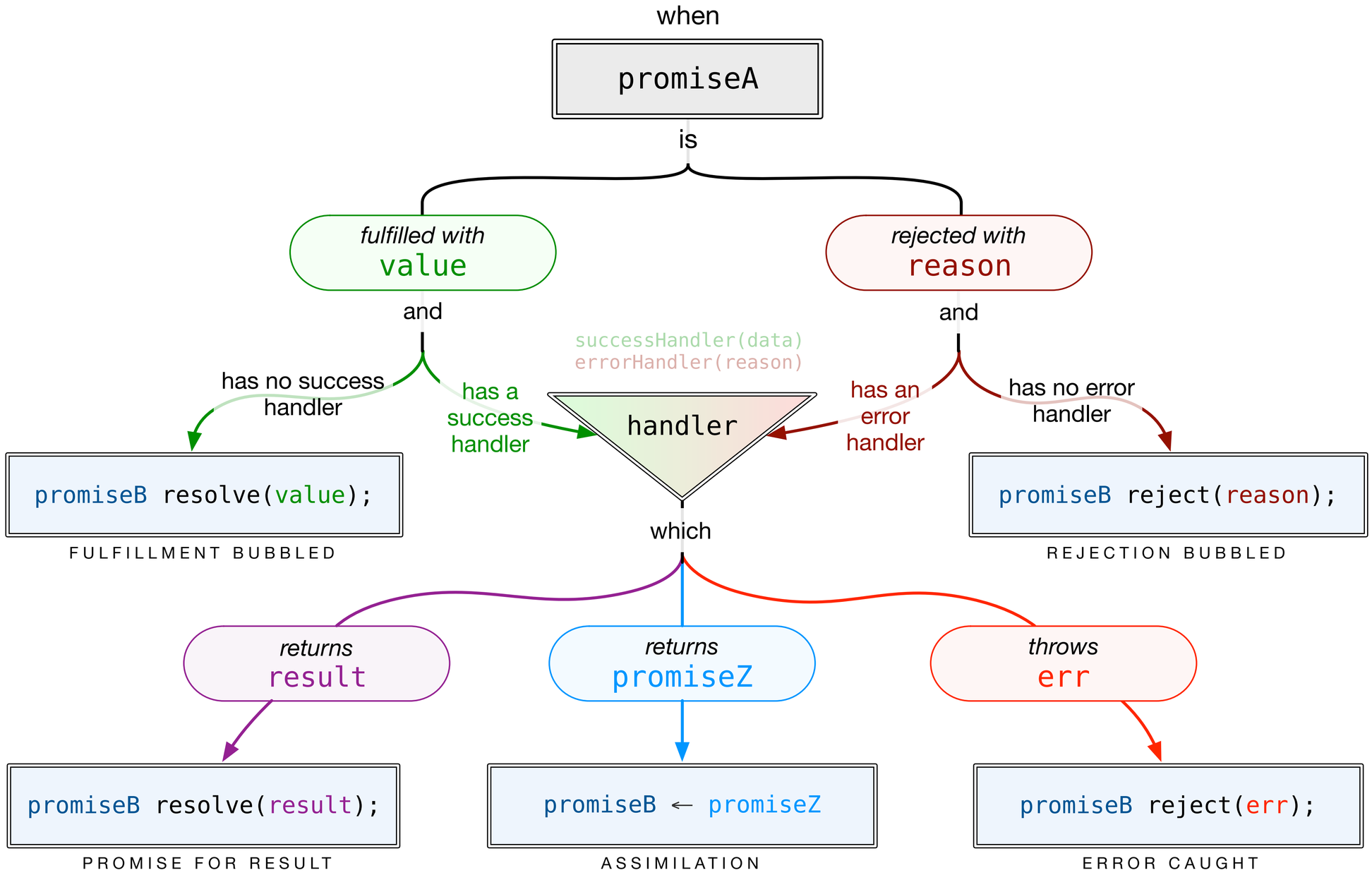


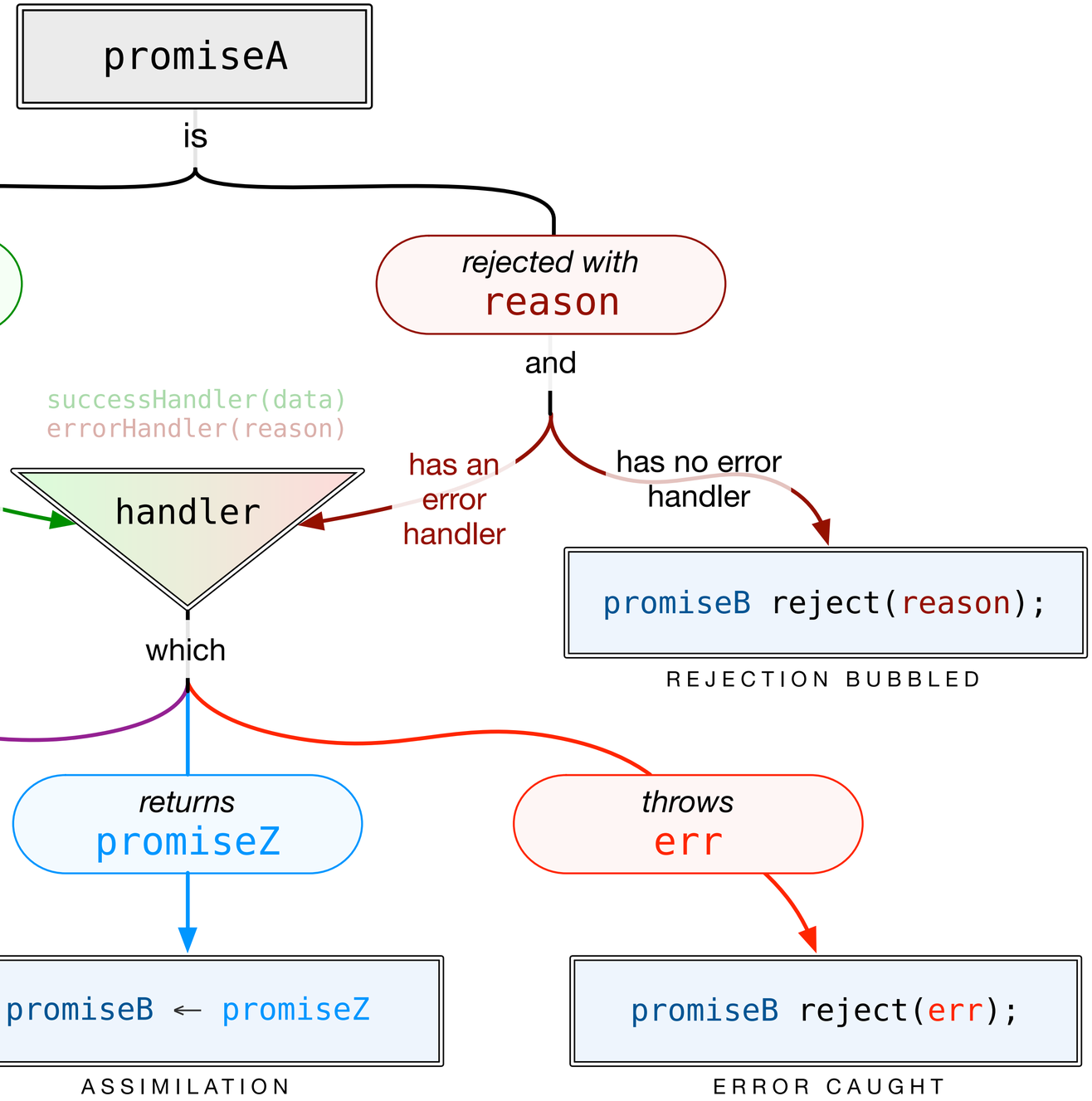
Review: Returning from Handler

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

Danger: Silent Errors?

- Since .then (also .catch) always returns a new promise, it never throws an error, but instead rejects the outgoing promise.
- This used to be a problem, but these days, promise libraries are “smart” and give warnings.
- In the future, unhandled rejections might throw errors!

Danger: Silent Errors

```
myPromise
.then(function (data) {
  use(data);
});
// no .catch !!!!!
```

—OR—

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

- Since .then (also .catch) always returns a new promise, it never throws an error, but instead rejects the outgoing promise
- Promise libraries are generally “smart” and give warnings

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);
  });
});
```

Converts an **array of promises** into a **promise for an array**

Node.js promises: native & Bluebird

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

```
npm install bluebird --save

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

You don't NEED bluebird
But you may want some extra goodies they give you:
`Promise.props`, `Promise.spread`, etc.

(some) Sequelize Promises

```
const usersPromise = User.findAll({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
- The original essay on promises (Liskov & Shrira)