Mechanics of Promises

Understanding JavaScript Promise Generation & Behavior

Async: continuation-passing

```
// Old/Imaginary
Page.findOne({where: {name: 'Promises'}}, function(err, page){
  if (err) return res.status(500).end();
  res.json(page);
});
```

Async: promise

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

Why?

Async: promise

```
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;
```

...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
Q.all( dayPromises ).then(function(days){
  res.render('calendar', {days: days});
});
```

Try... old-school callback hell?!

```
var days = [];
// Old/Imaginary
Day.findOne({where: {dayNum: 1}}, function(err, day1){
  if (err) return res.status(500).end();
  days.push(day1);
   Day.findOne({where: {dayNum: 2}}, function(err, day2){
      if (err) return res.status(500).end();
      days.push(day2);
      Day.findOne({where: {dayNum: 3}}, function(err, day3){
          if (err) return res.status(500).end();
          days.push(day3);
          Day.findOne({where: {dayNum: 4}}, function(err, day4){
             if (err) return res.status(500).end();
             days.push(day4);
             Day.findOne({where: {dayNum: 5}}, function(err, day5){
                 if (err) return res.status(500).end();
                days.push(day5);
                 Day.findOne({where: {dayNum: 6}}, function(err, day6){
                    if (err) return res.status(500).end();
                    days.push(day6);
                    Day.findOne({where: {dayNum: 7}}, function(err, day7){
                        if (err) return res.status(500).end();
                        days.push(day7);
                        res.render('calendar', {days: days});
                    })
                })
             })
          })
      })
   })
});
```

Promise Chaining!

```
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)
iHDWALLPAPERS.COM

Promises are Objects

state (pending, fulfilled, or rejected)

information (value or a reason)

} (hidden if possible)

.then()

(public property)

Promise 1
State: pending
Data: undefined

Fulfilled with

`value`

Promise 1
State: fulfilled
Data: `value`

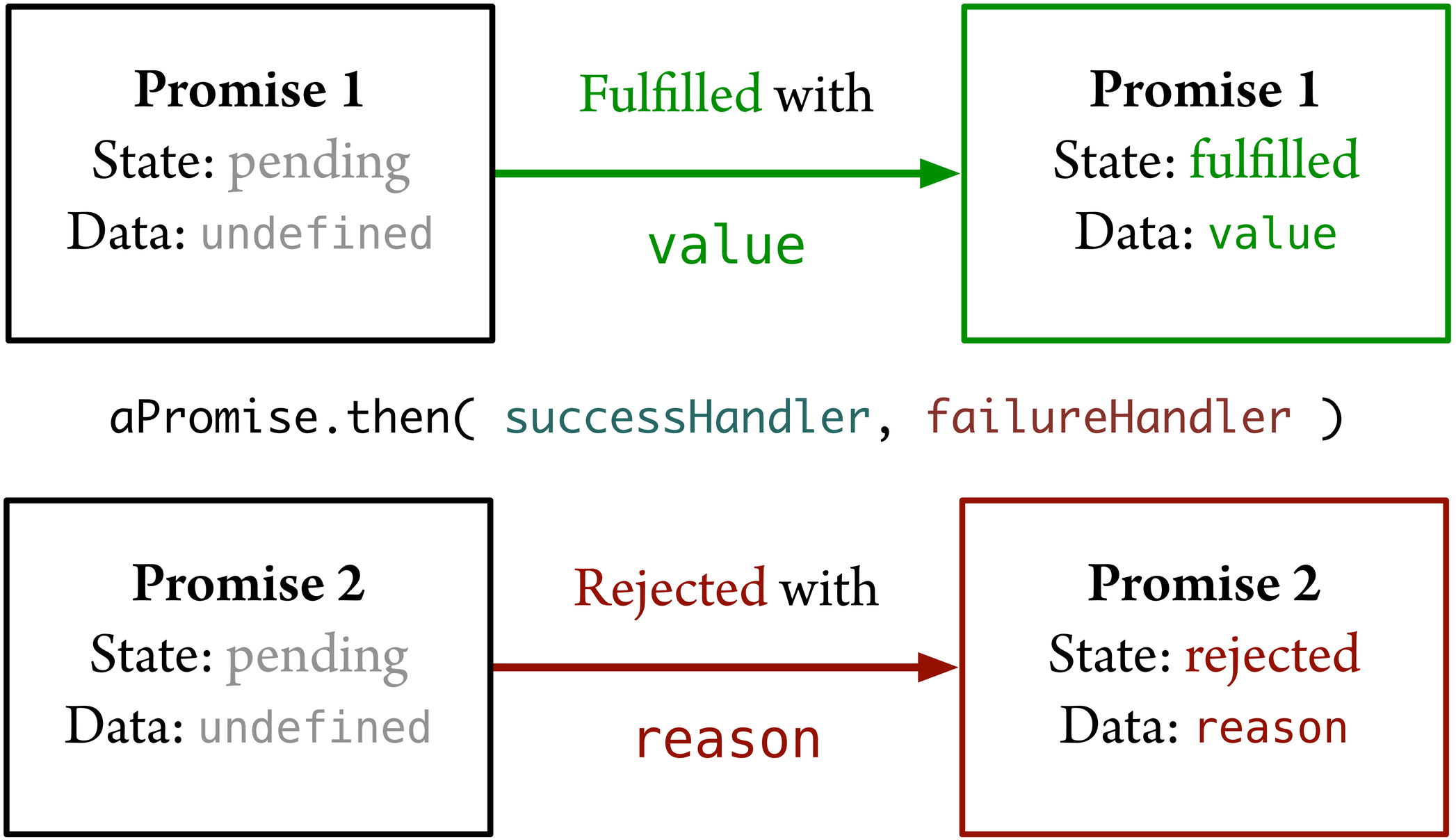
promises only change stat

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
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

OR

1. **promise settles**
2. *add handler*
3. handler is called once

```
var userPromise = User.find({ where: { name: ‘Kate’ })
```

```
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
});
```

Enter 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] );
    });
  });
});
```

Pyramid of DOOOOOOOOM

Callback Hell... with error handling!

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

Promise-Land

```
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 can literally not return from a typical async callback

Are we sure it is a black-hole?!

```
// This will not work because ^^ black hole
var person = asyncGetGroup( 123, function got( group ) {
  return group.users[0];
});
```

Is that light I see?

```
// This might work (but not reliable) – race condition
var person;
asyncGetGroup( 123, function got( group ) {
  person = group.users[0];
});

// later in code
var headline = person.name; // might be undefined!
```

Fantasy/Escape from Darkness

```
var person;
asyncGetData( function got( data ) {
  person.save( data ); // once async completes
});

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

That's a promise

Complex landscape… but standardized

- **Multiple proposed standards · CommonJS**
- **One leading standard · Promises/A+**
  - Includes native ES6 promises

- **Two different approaches for generating new promises**
  - CommonJS-style *deferrals* (one extra entity)
  - ES6-style *constructors* (simplified)

jQuery 3 now A+!

- **jQuery gurus beware! `$.Deferred` differed from current standards and was considered flawed.**

So where do real promises come from?

- **Existing libraries may return promises**
  - Angular `$http`
  - Sequelize queries / db actions
- **Wrap async calls in promise-makers**
  - Q / `$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 promiseForTxt = new Promise( function (resolve, reject) {
  fs.readFile('log.txt', 'utf-8', function (err, text) {
    if (err) reject( err );
    else resolve( text );
  });
});

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

Constructor-style promise generation

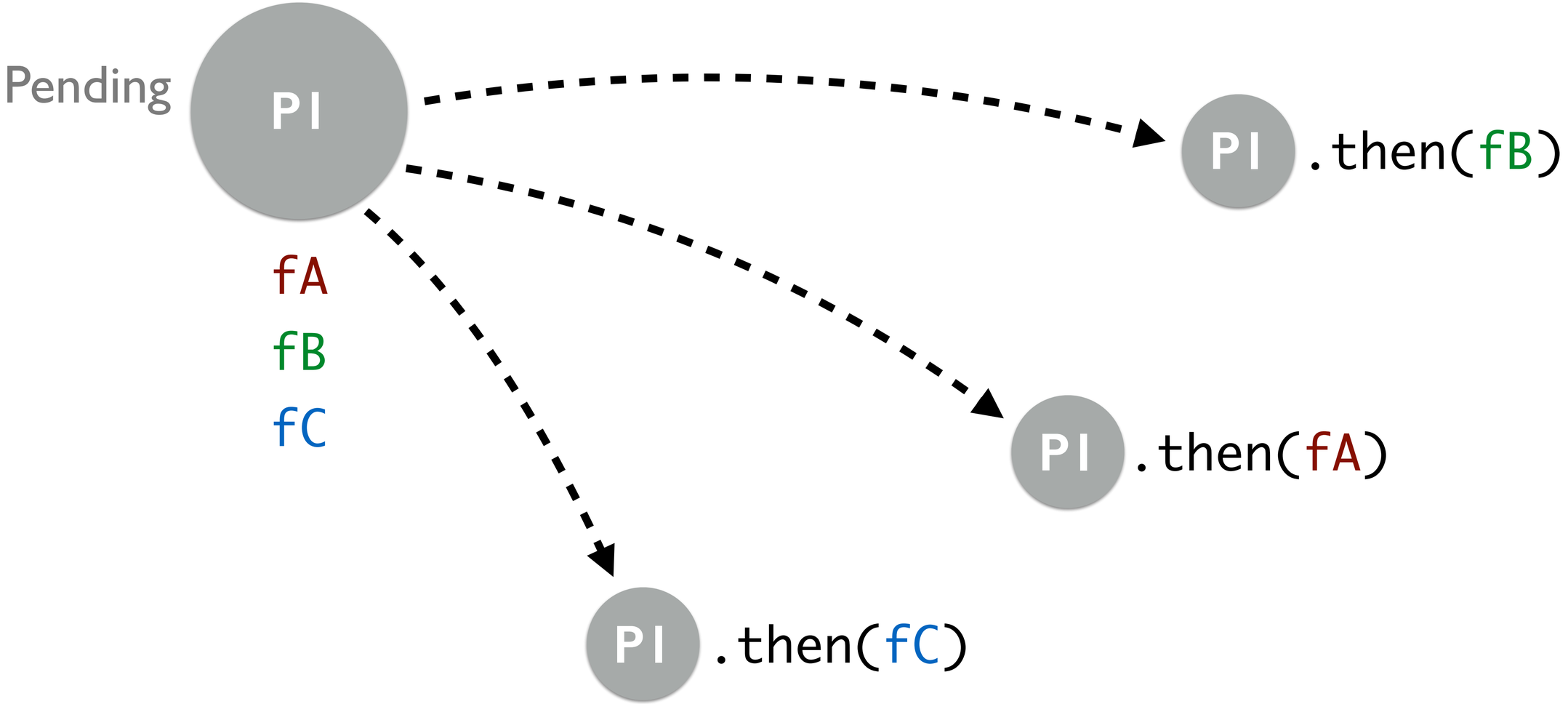
```
var promiseForTxt = function (filename, enc){
  return new Promise(function (fulfill, reject){
    fs.readFile(filename, enc, function (err, res){
      if (err) reject(err);
      else fulfill(res);
    });
  });
}
// elsewhere
promiseForTxt(filename, enc).then(someSuccessHandler, someErrorHandler);
```

Deferral-style promise generation

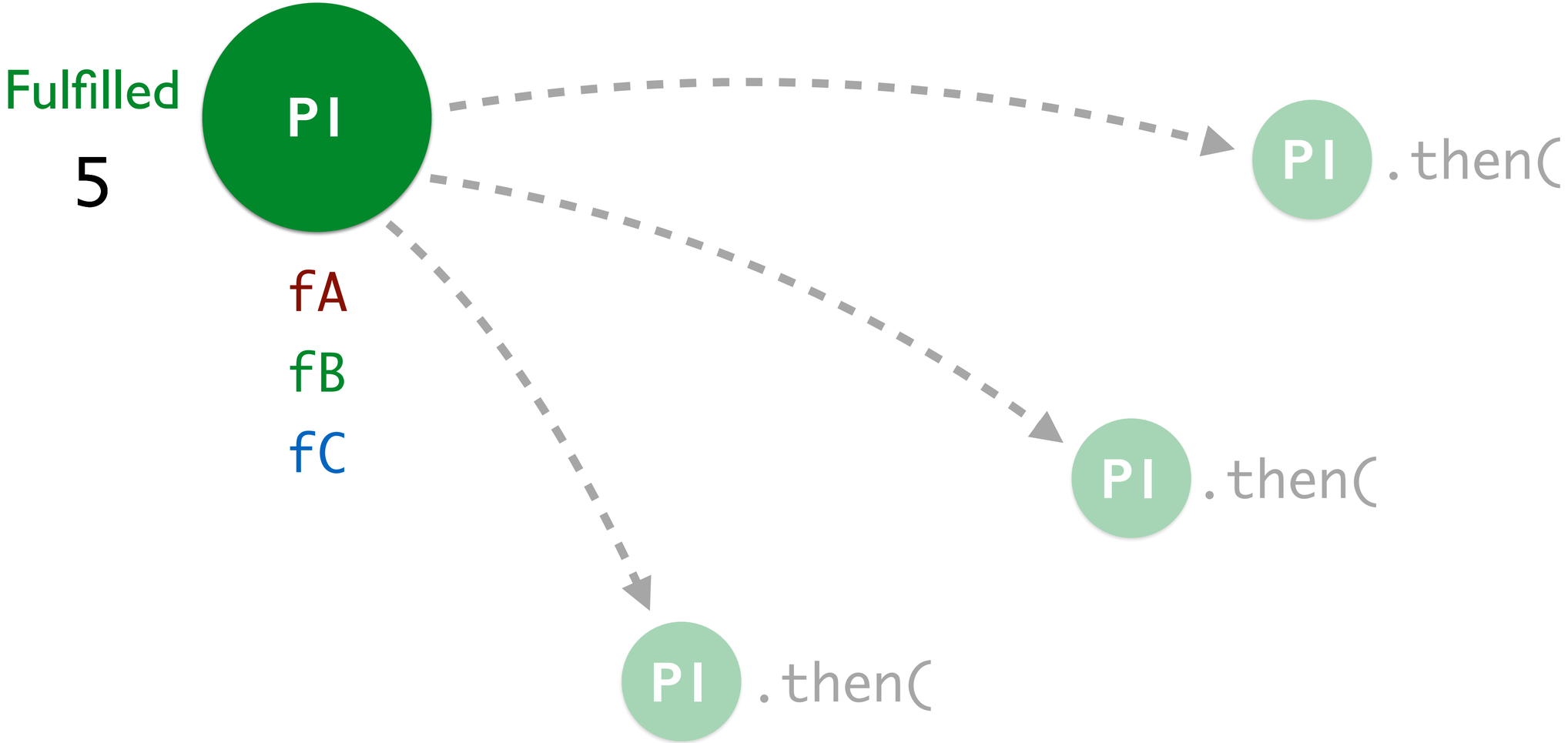
```
var myDeferral = $q.defer();
someAsyncCall( function (err, data) {
  if (err) myDeferral.reject( err );
  else myDeferral.resolve( data );
});
var myPromise = myDeferral.promise;

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

(Break)

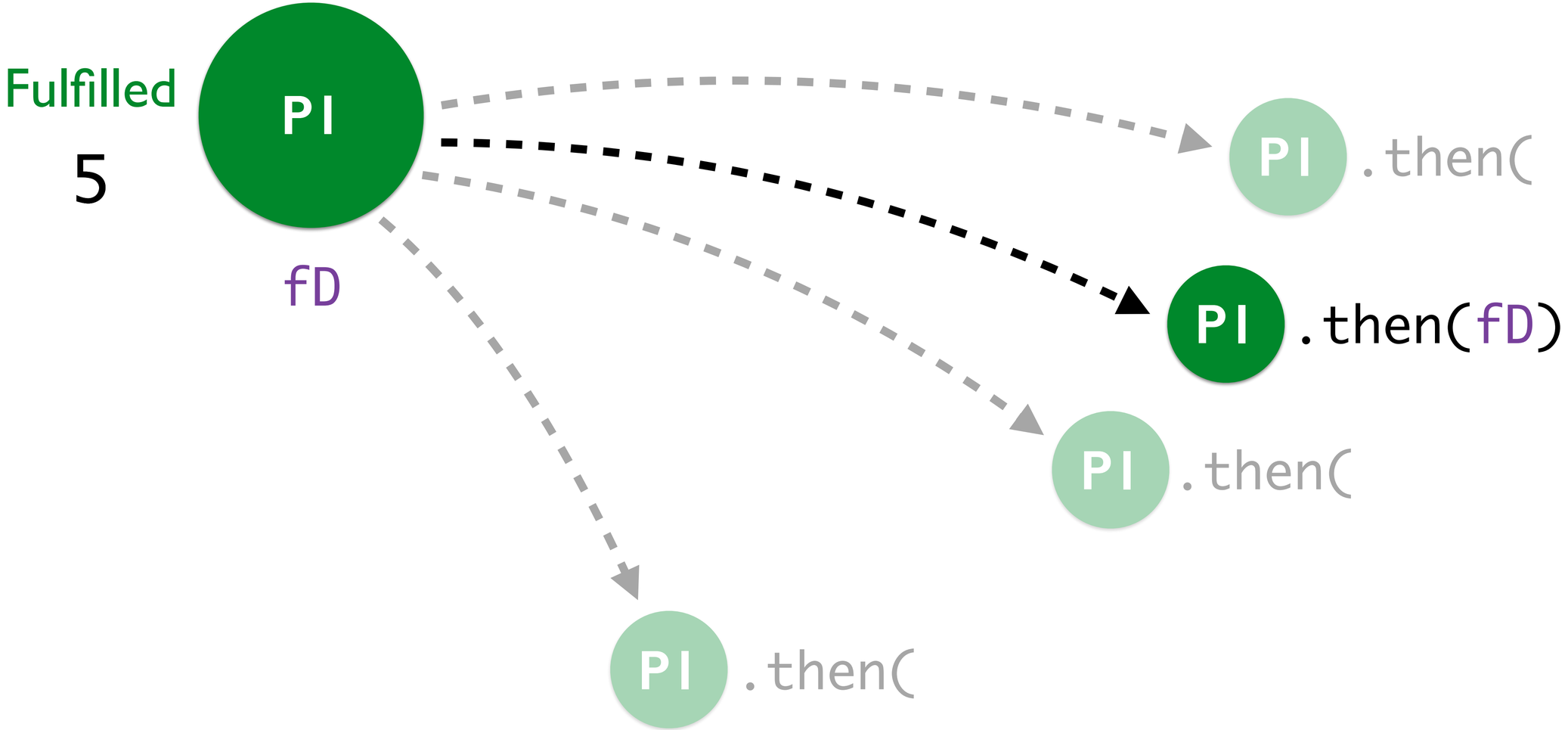
.then on same promise (not chaining!)



.then on same promise (not chaining!)



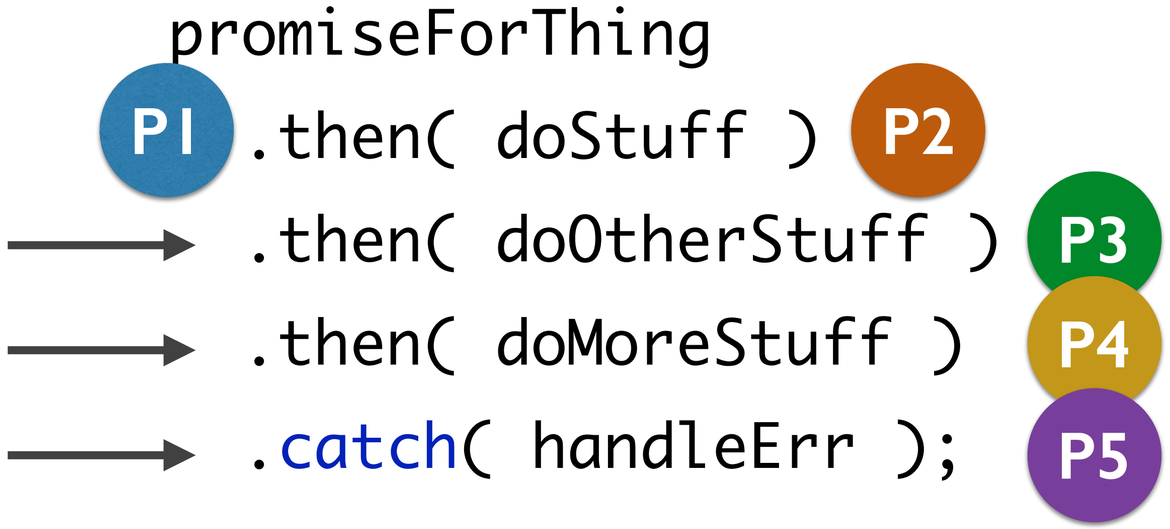
.then on same promise (not chaining!)



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)

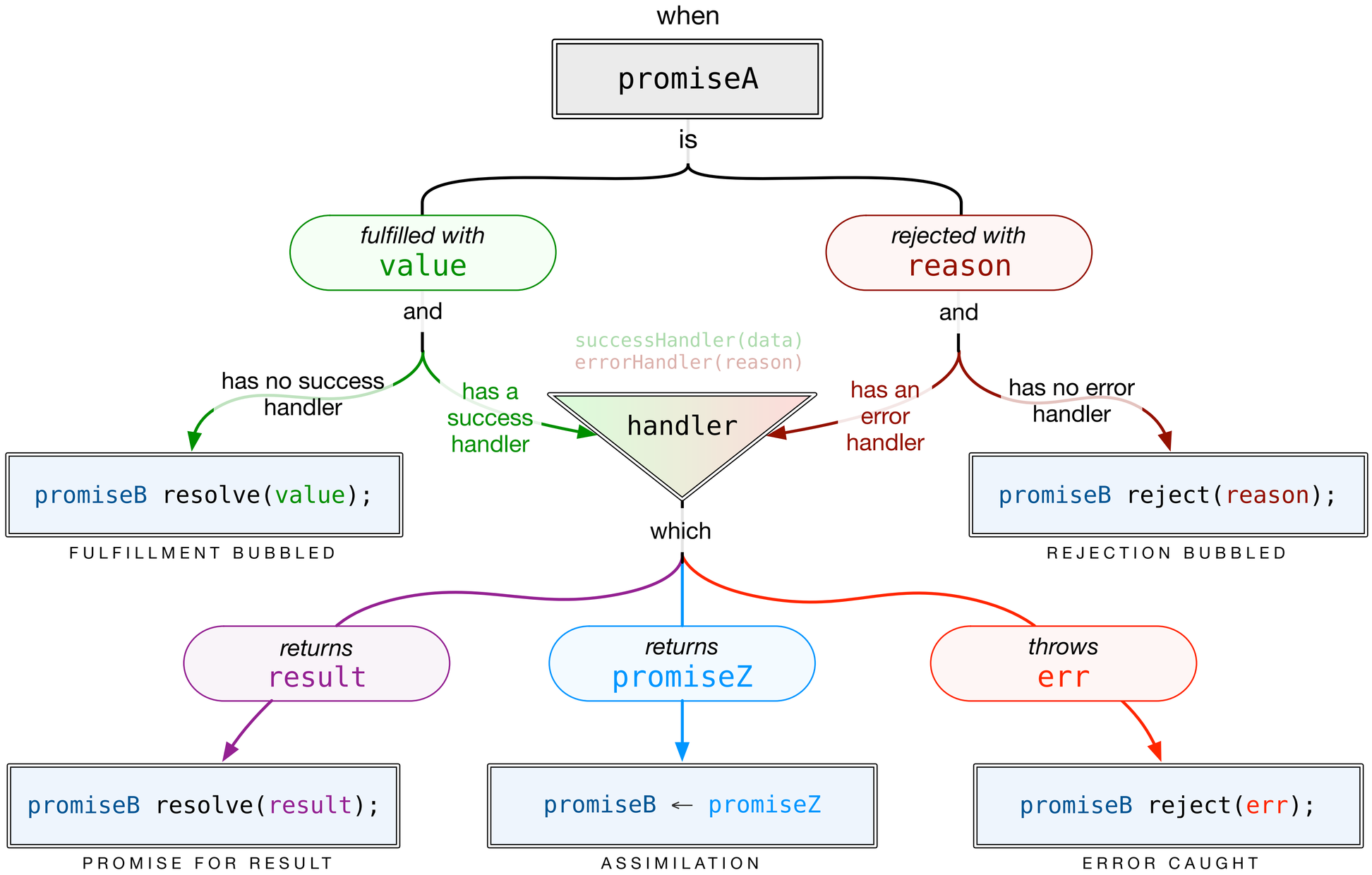
And why we can return from a handler

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

What is promiseB a promise for?

Brace yourselves...

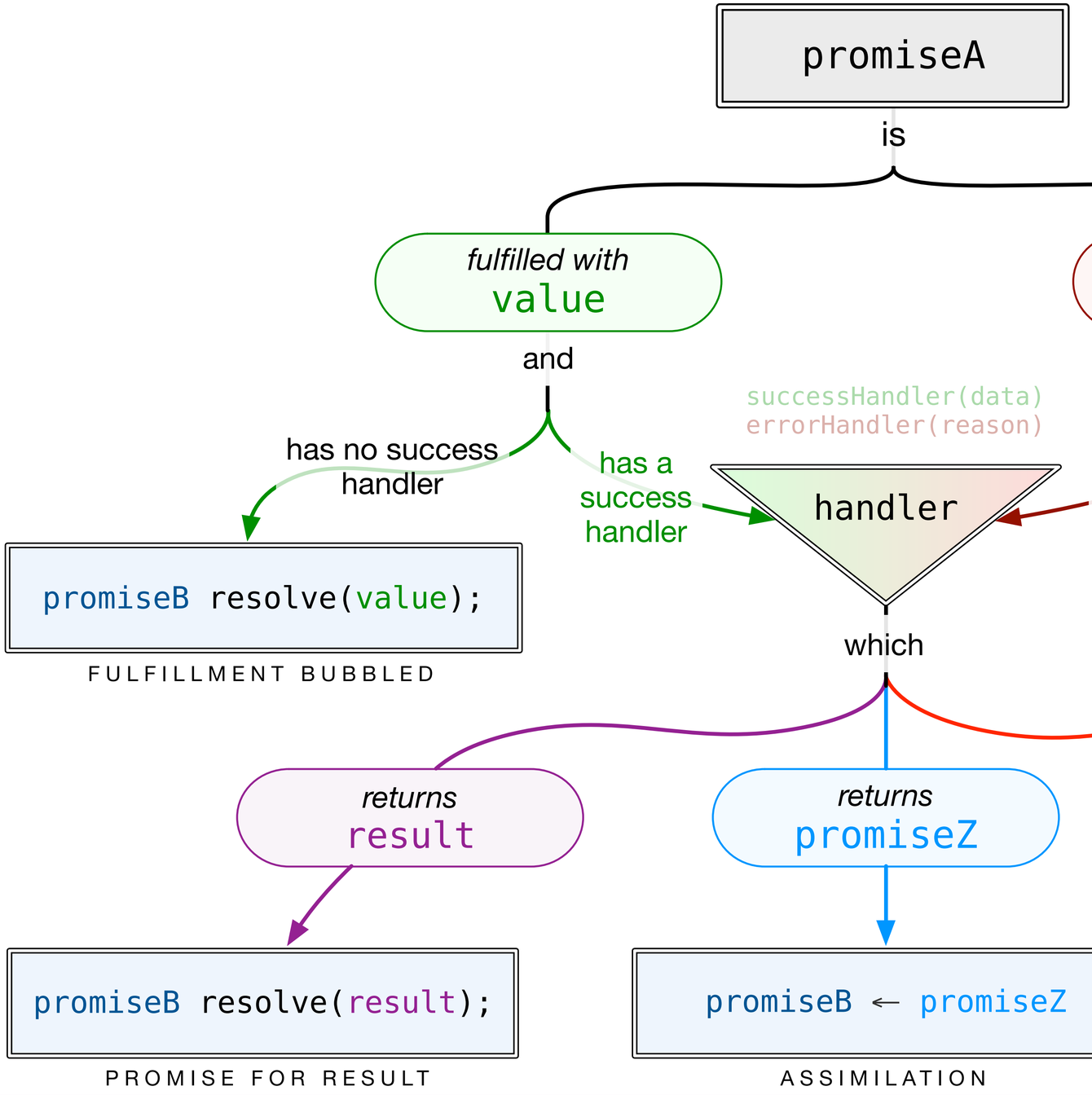
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



```
// 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:

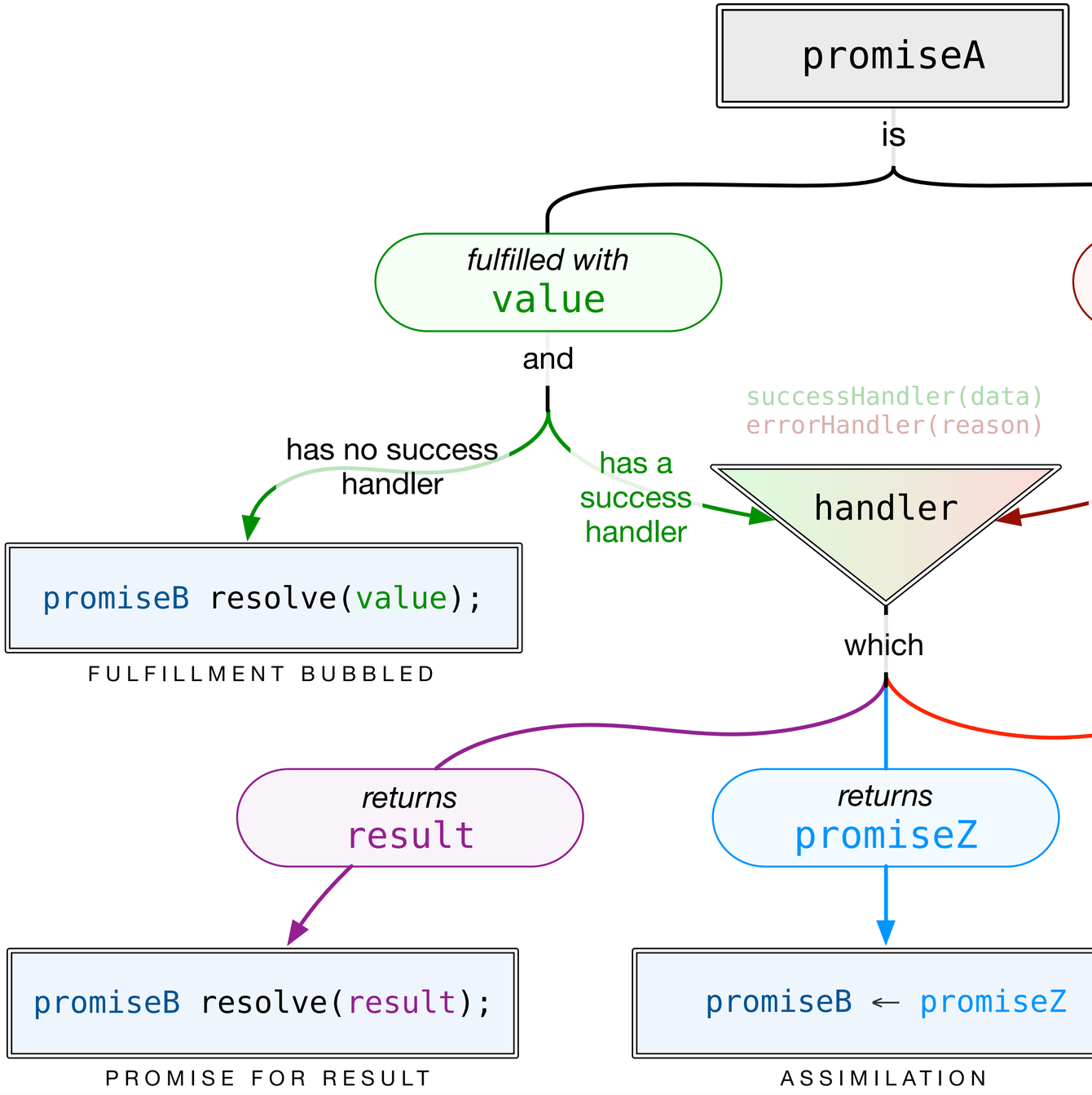
Console log reads “Hello.”

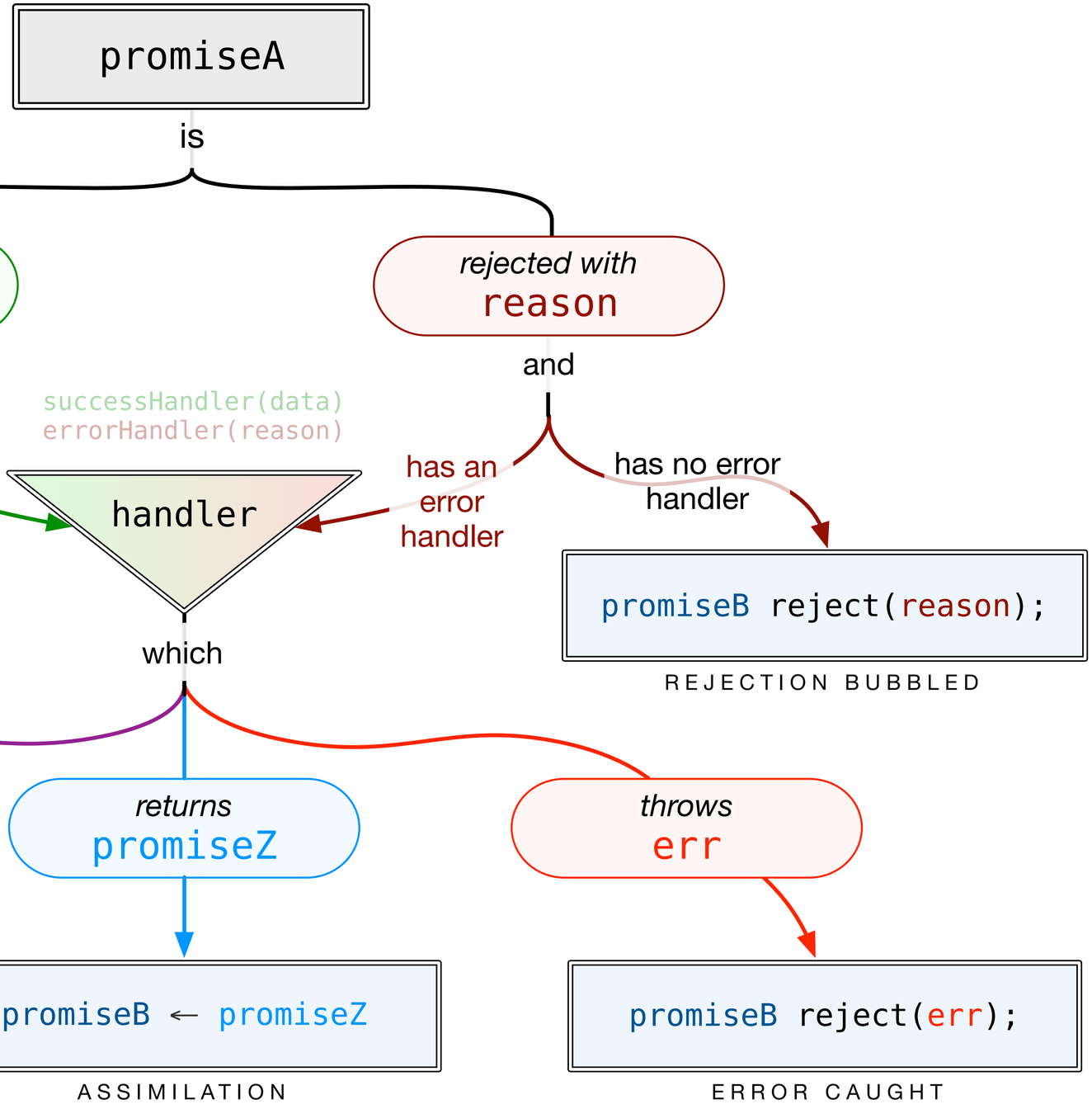


```
// promise0 fulfills with 'Hello.'

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

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

Console log reads “Hello.”

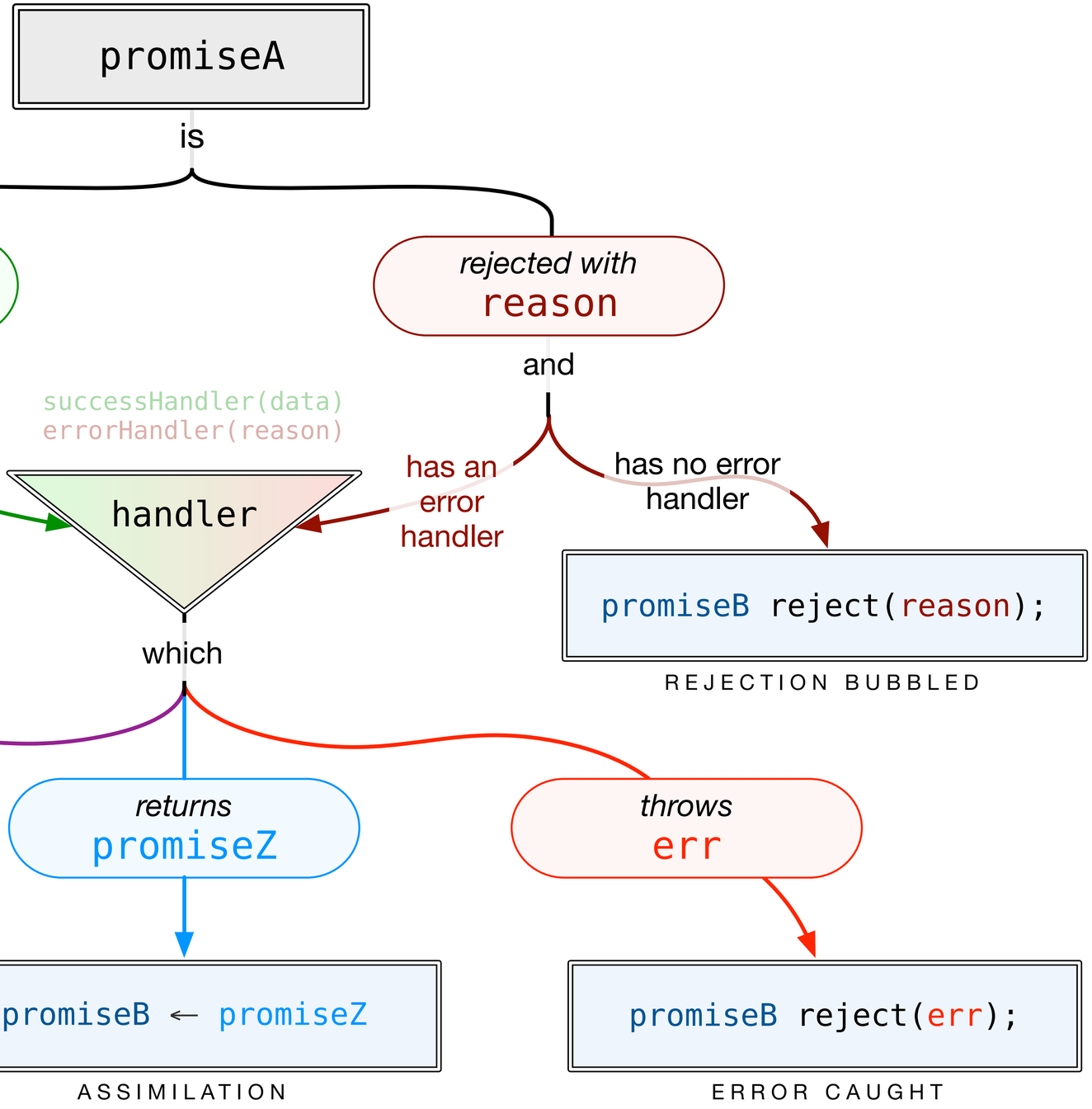





```
// promise0 rejected with 'Sorry'

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".



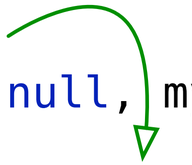

```
function logYell (input) {
  console.log(input+'!');
}

// promise0 rejected with 'Sorry'

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

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

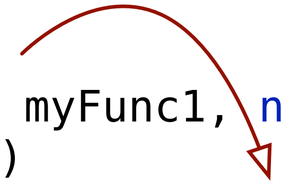
Console log is "Sorry!"

Review: Success & Error Bubbling

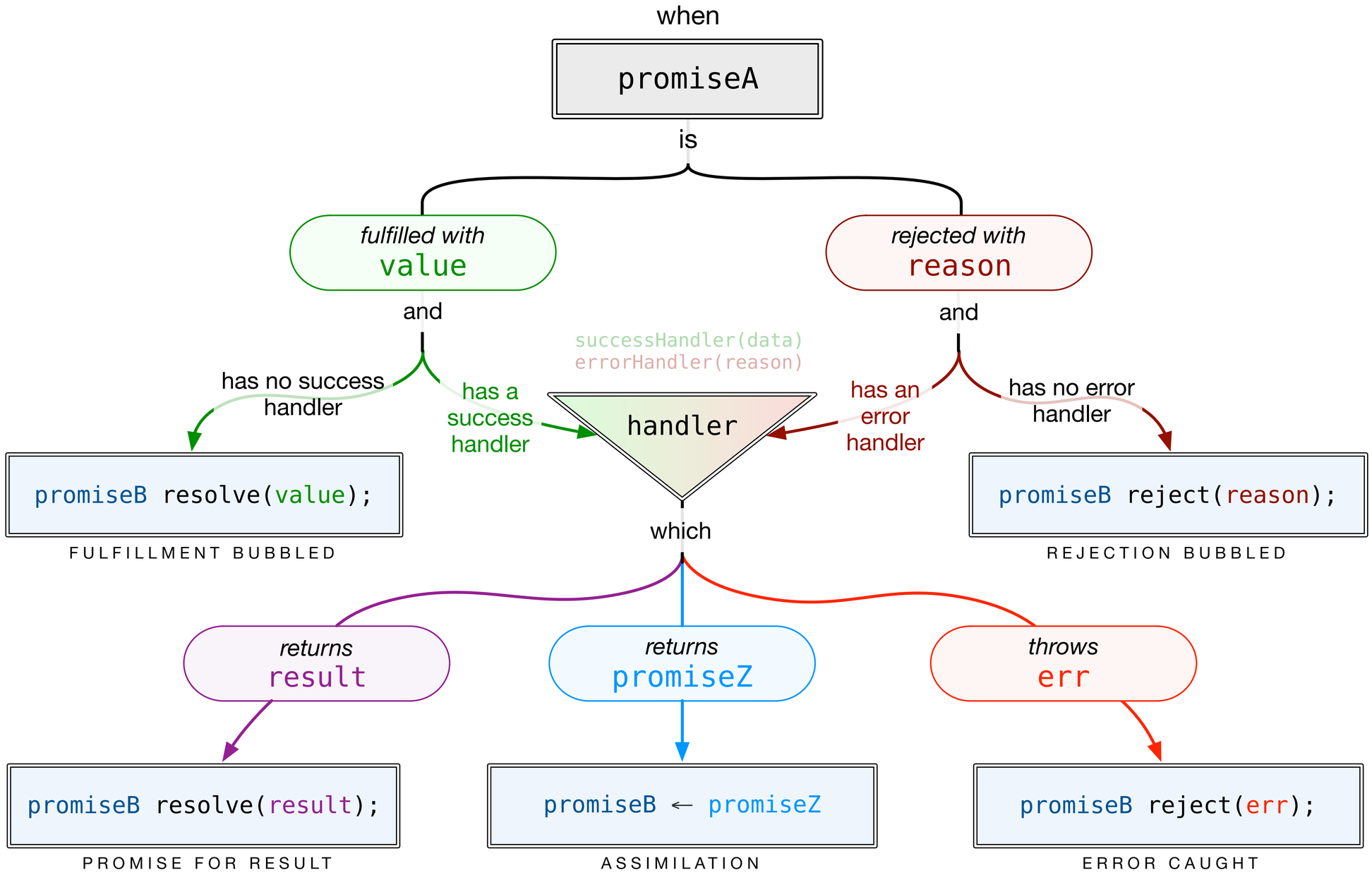
```
// promiseA is fulfilled with 'hello'

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

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



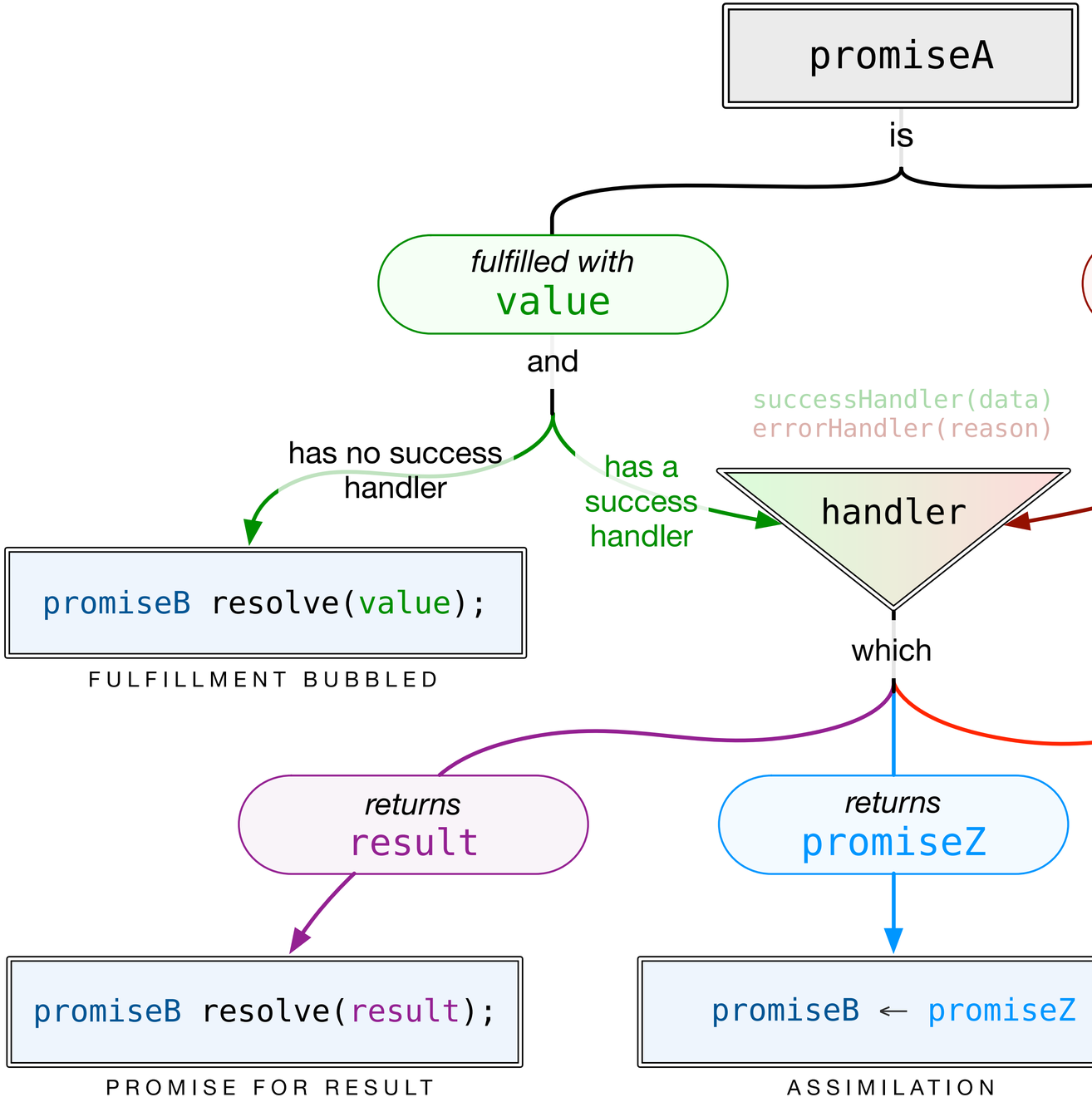
```
// promiseA is rejected with 'bad request'

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

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



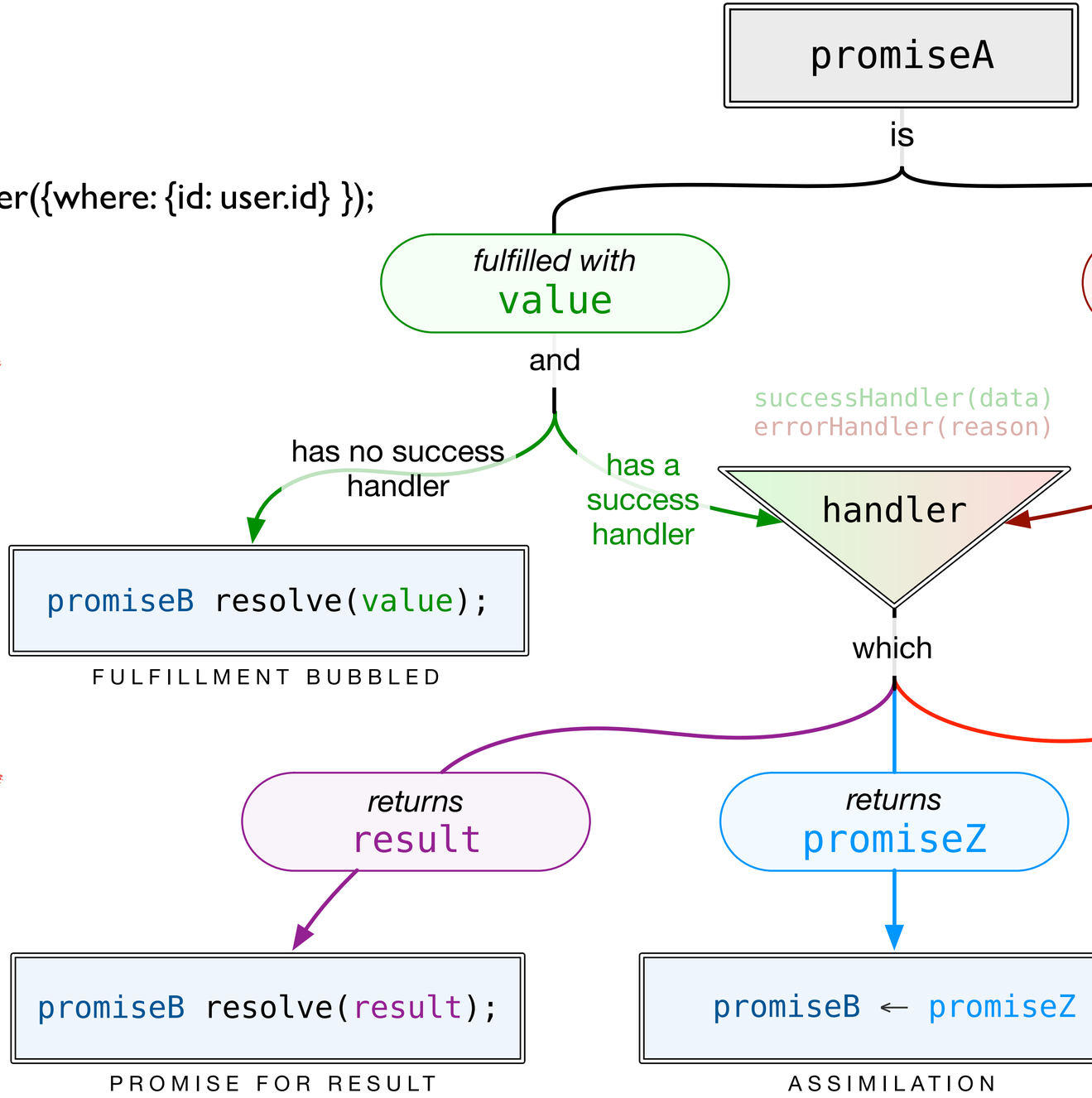
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



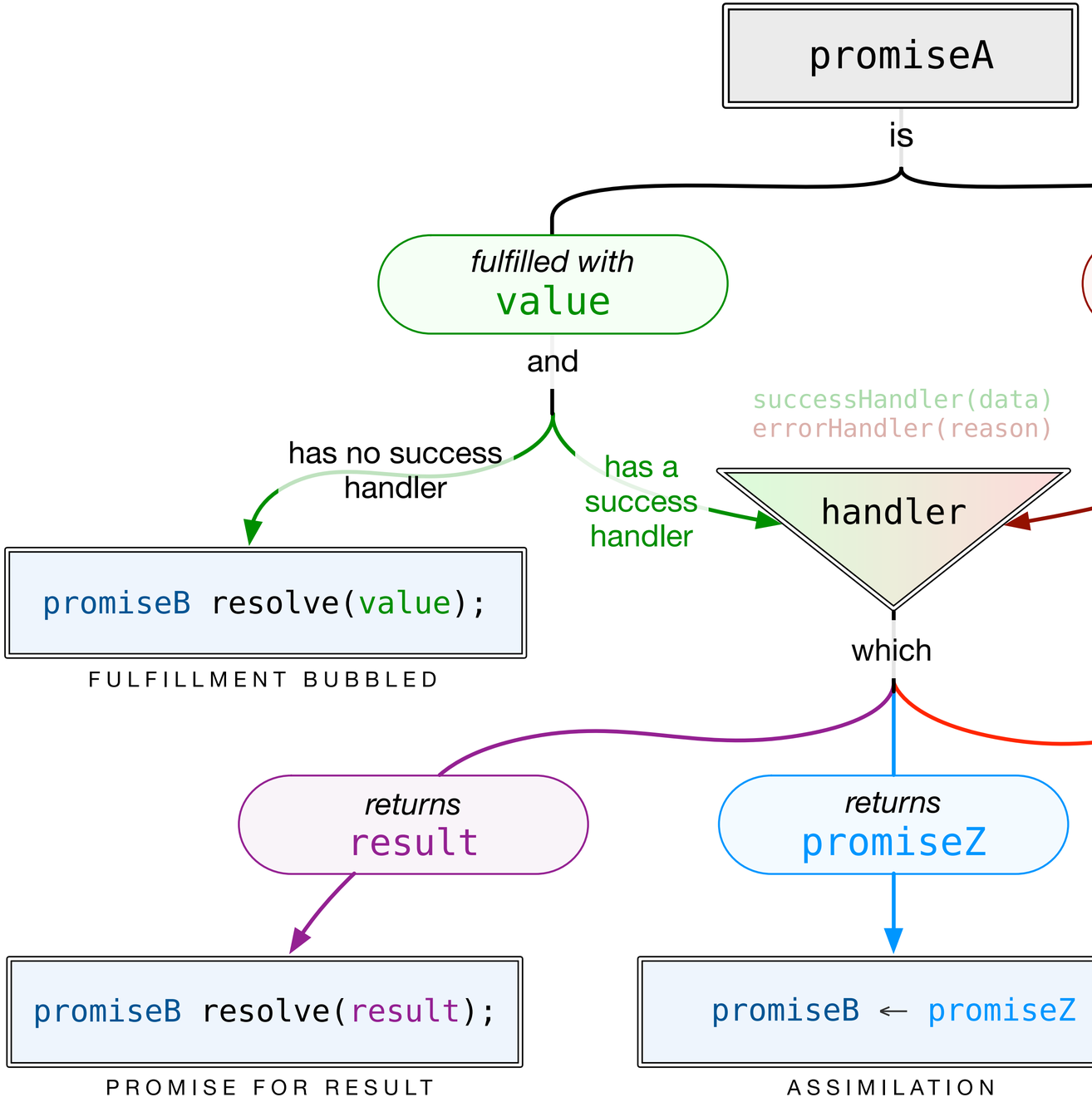
```
// output promise is for returned val

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 );
  });
```



```
promiseForUser.then(function (user) {
    var promiseForMessages = DBLibrary.findAllMessagesForUser({where: {id: user.id} });
    return promiseForMessages;
})
.then(function (promisesForMessages) { //based on previous logic
    promisesForMessages.then(function (messages) {
        console.log(messages);
    })
  //this ^^^ is nesting thens and is the same as the callback hell
})
```



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

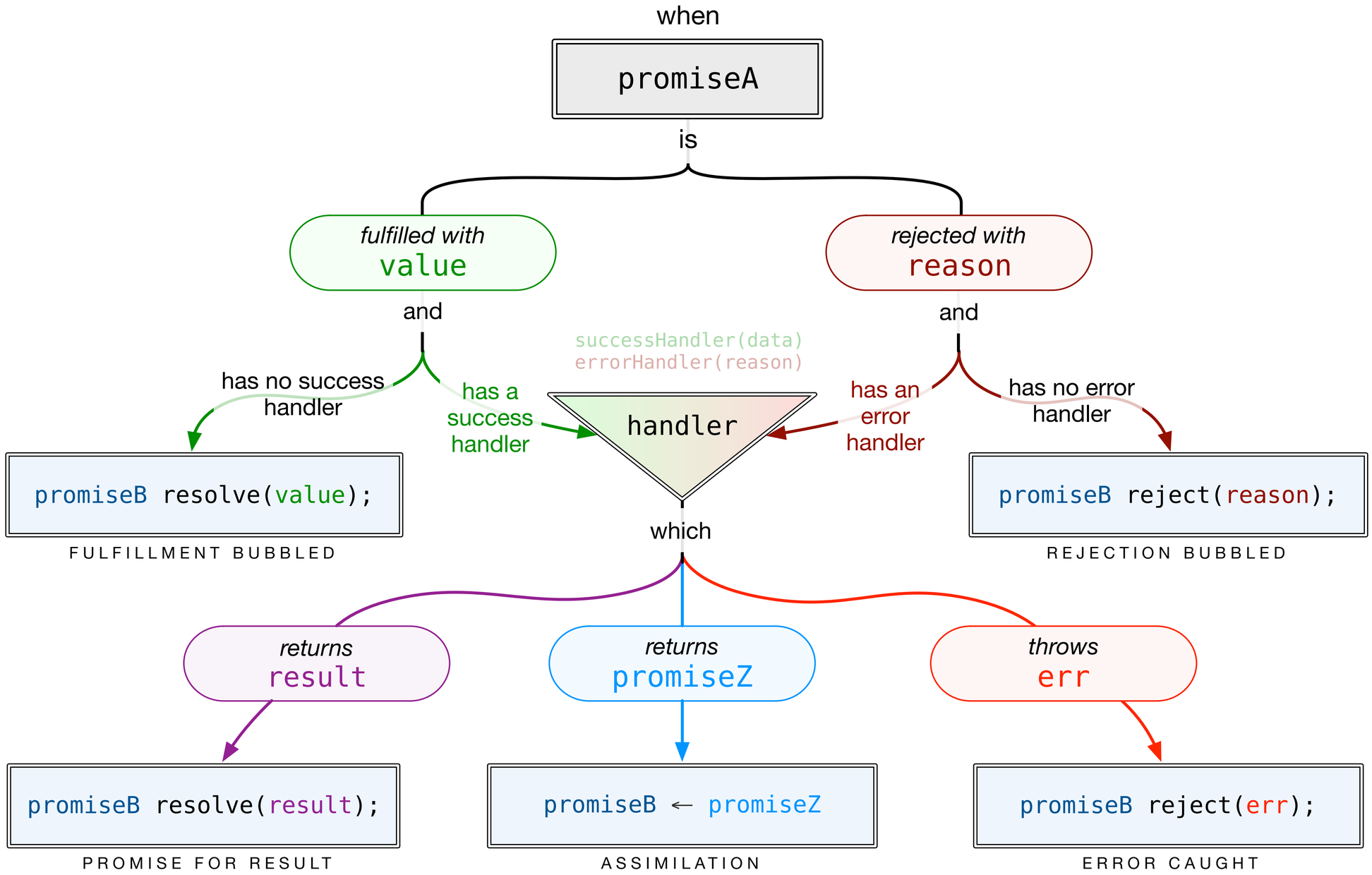


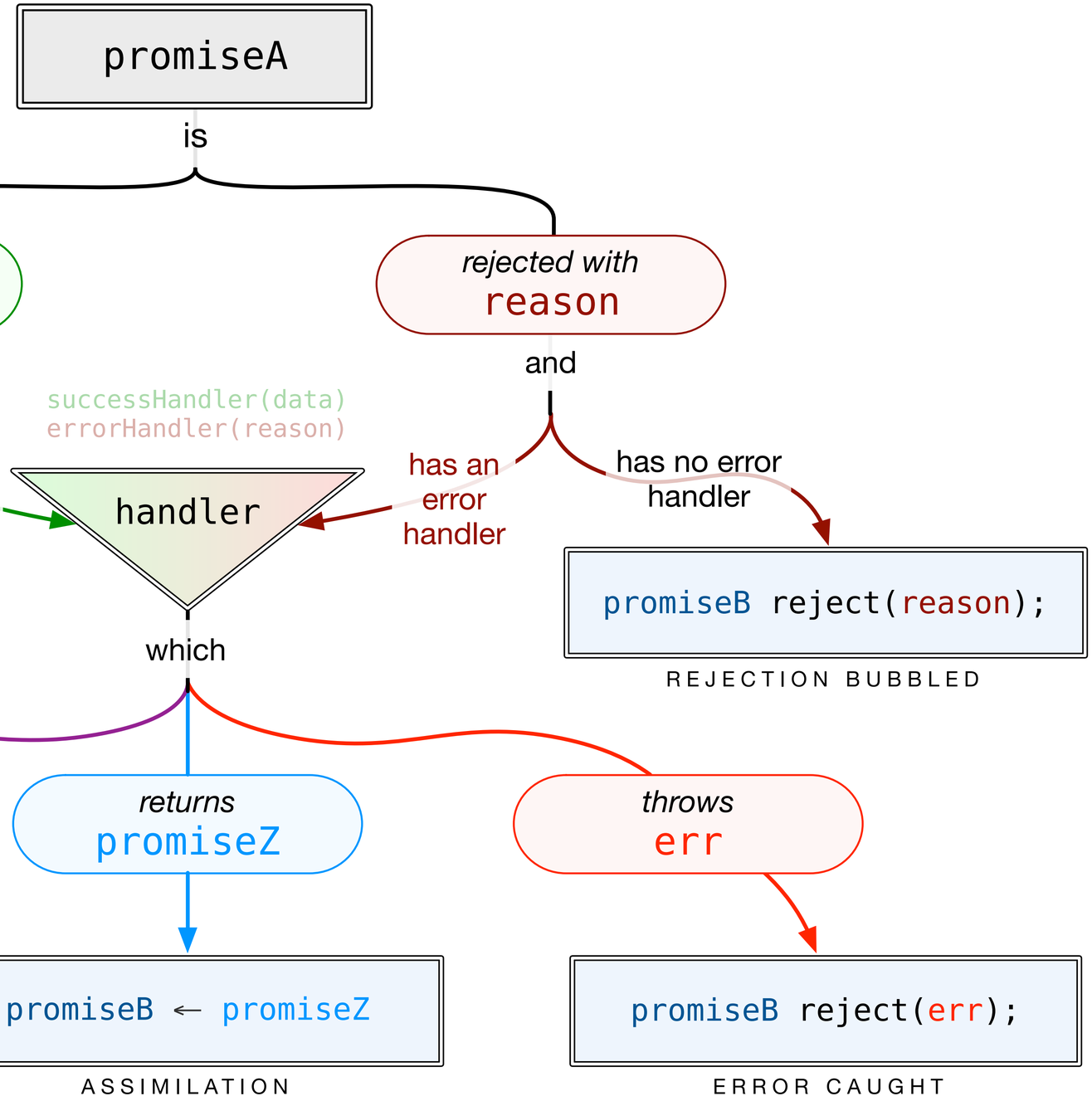
Review: Returning from Handler

```
// 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();
```

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

Node.js promise libraries: Q & Bluebird

```
npm install q --save

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

```
npm install bluebird --save

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

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