Mechanics of Promises

Understanding JavaScript Promise Generation & Behavior

TRAJECTORY

- Review: Why Promises?
- Using the Promise constructor
- What is a Promise (like, really)?
- A simple Promise implementation
- Understanding .then

Continuation Passing

```
fs.readFile('./foo.md', (err, data) => {
  if (err) handle(err)
  else doSomethingWith(data)
})
```

Promises

```
promisifiedReadFile('./foo.md')
  .then(
    function (data) { doSomethingWith(data) },
    function (err) { handle(err) }
  )
```

Why?

Callback Hell

deep, confusing nesting & forced, repetitive error handling

```
// Basic async callback pattern.

getUserData(userId, function (err, data) {
  console.log(data)
})
```

```
// Callback Hell

getUserData(userID, function (userData) {
  getMessage(userData.messageIDs[0], function (message) {
    getComments(message, function (comments) {
      console.log(comments[0])
    })
  })
})
```

```
// AAAAAAAAAAHHHHHHHHHH

getUserData(userID, function (err, userData) {
  if (err) console.log('user fetch err: ', err)
  else getMessage(userData.messageIDs[0], function (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)
  })
```

"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"

Break free from the async call!

```
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));
```

The Promise Constructor

```
function promisifiedReadFile (fileName) {
  // let's write me!
}
```

```
function promisifiedReadFile (fileName) {
  return new Promise(function (resolve, reject) {

  })
}
```

```
function promisifiedReadFile (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function (err, data) {



   })
  })
}
```

```
function promisifiedReadFile (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function (err, data) {
      if (err) reject(err)

    })
  })
}
```

```
function promisifiedReadFile (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function (err, data) {
      if (err) reject(err)
      else resolve(data)
    })
  })
}
```

So, what is a promise?

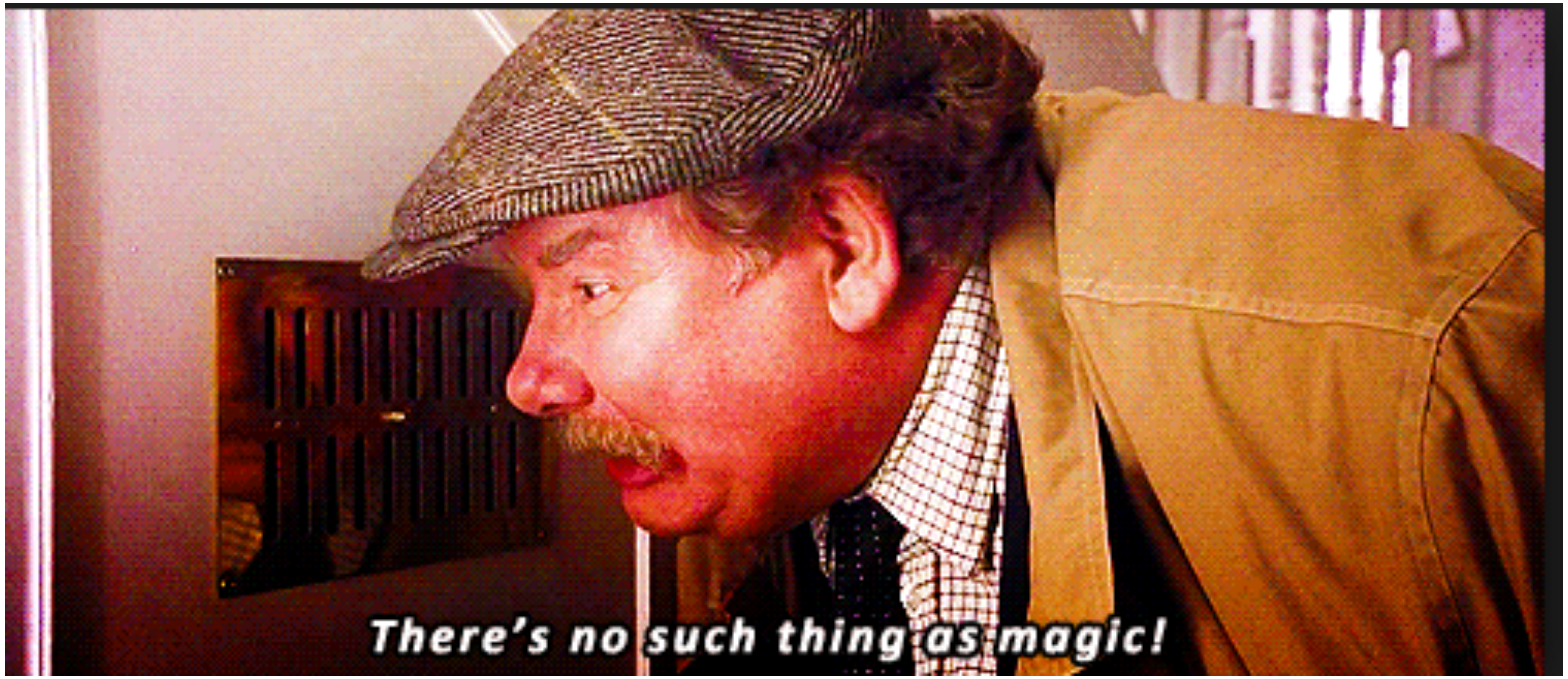
“A promise represents the eventual result of an asynchronous operation.”

— THE PROMISES/A+ SPEC

magic!

(no)

iHDWALLPAPERS.COM


There's no such thing as magic!

Promises are Objects

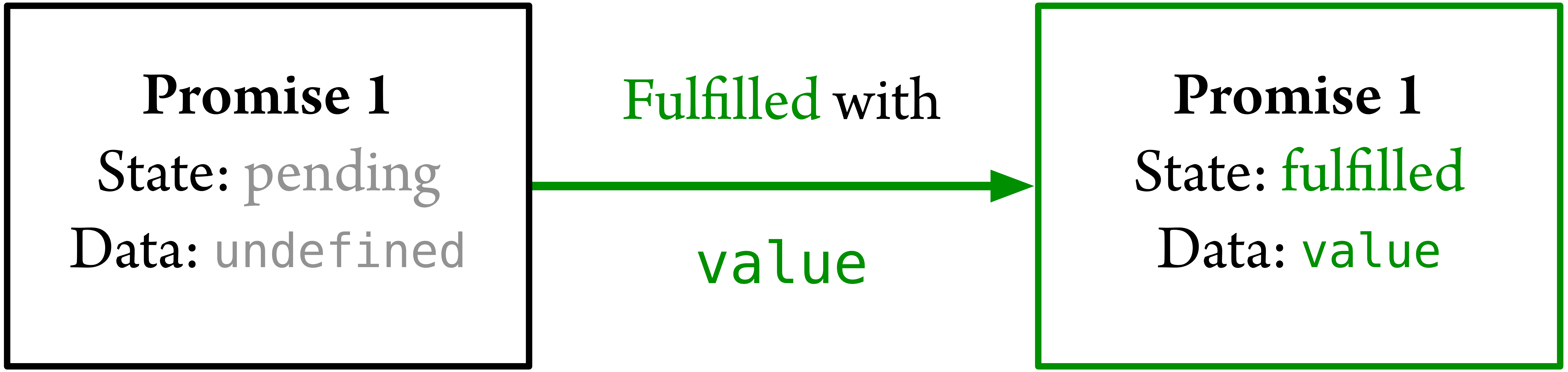
state (pending, fulfilled, or rejected)

information (value or a reason)

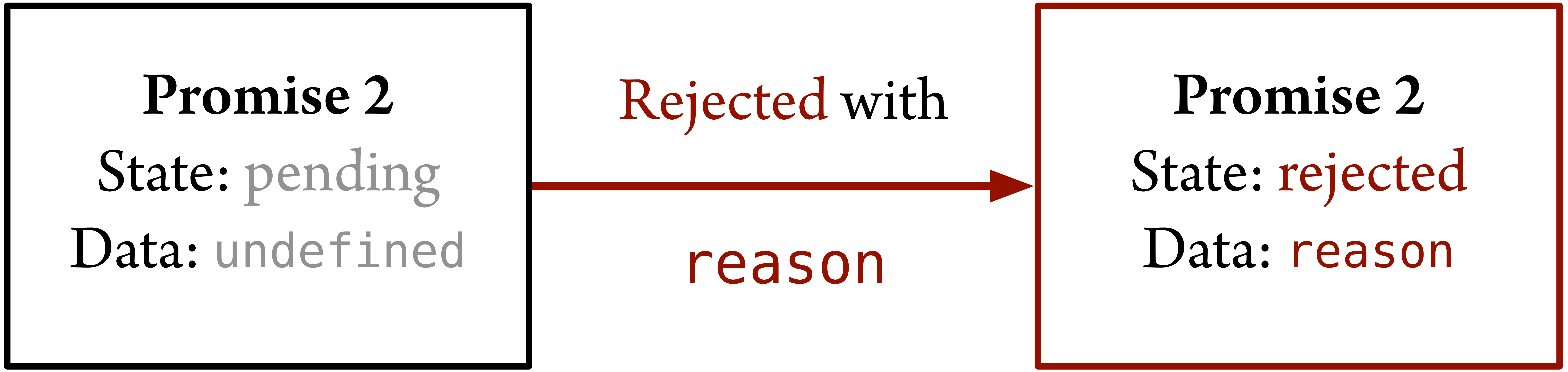
.then()

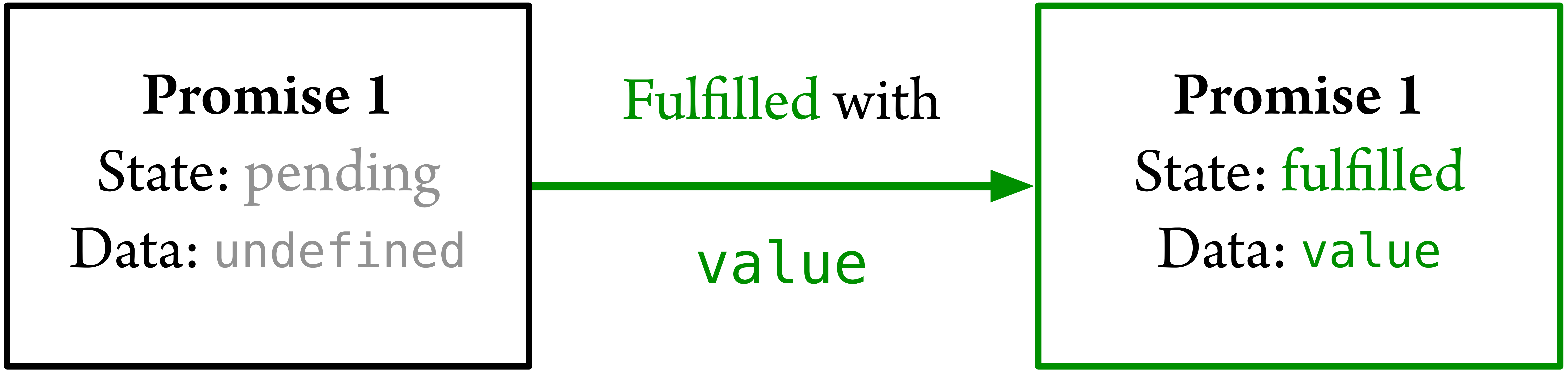
} (hidden if possible)

(public property)

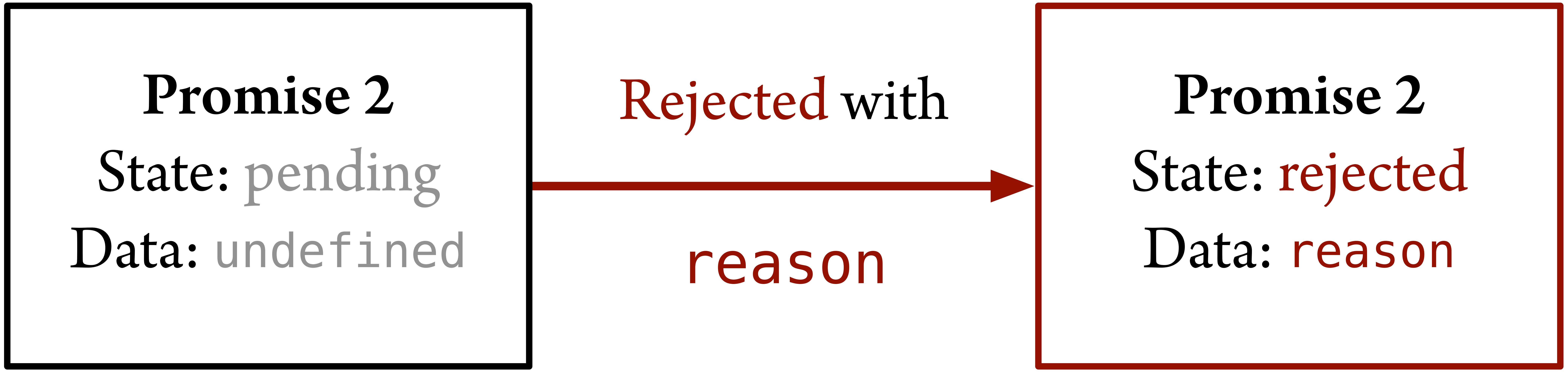


promises only change state while pending





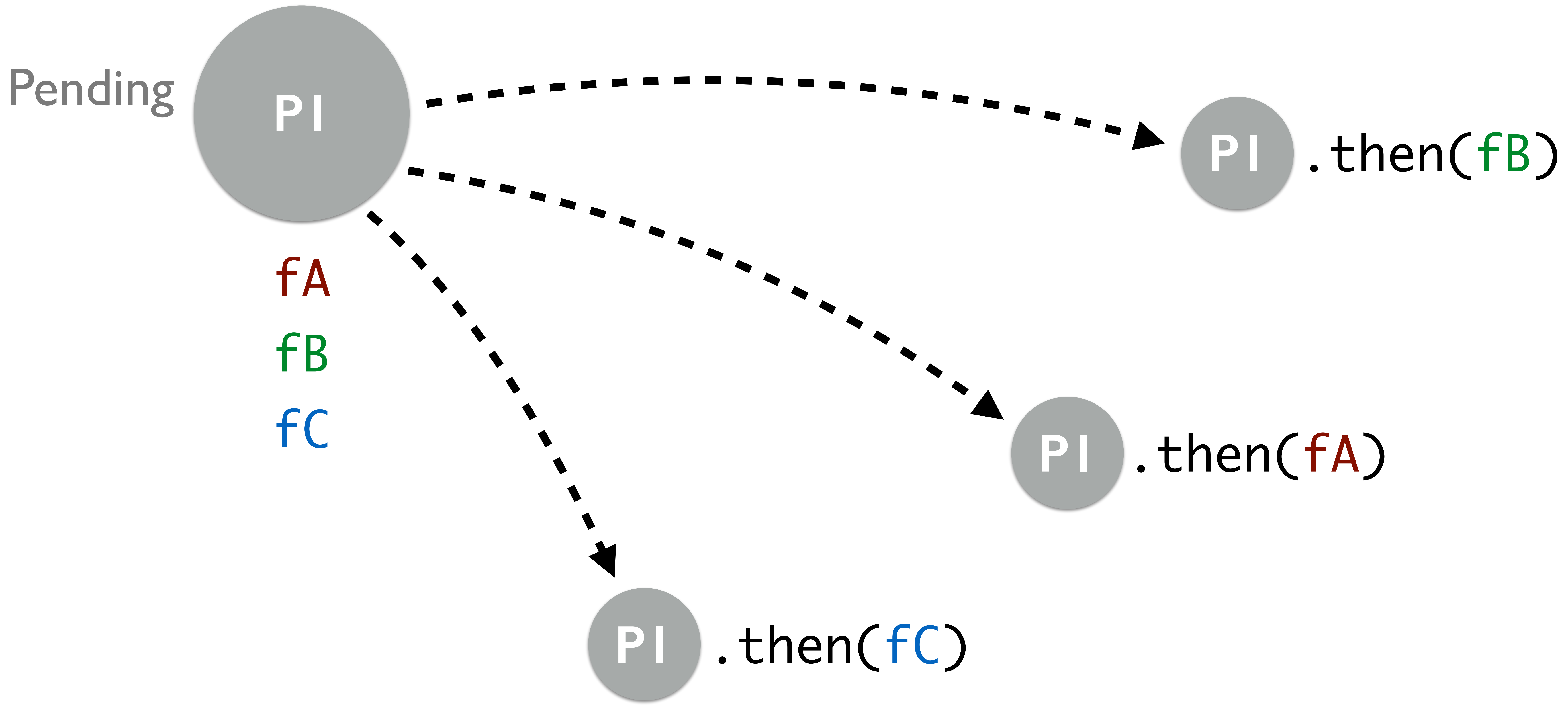
```
aPromise.then( successHandler, failureHandler )
```



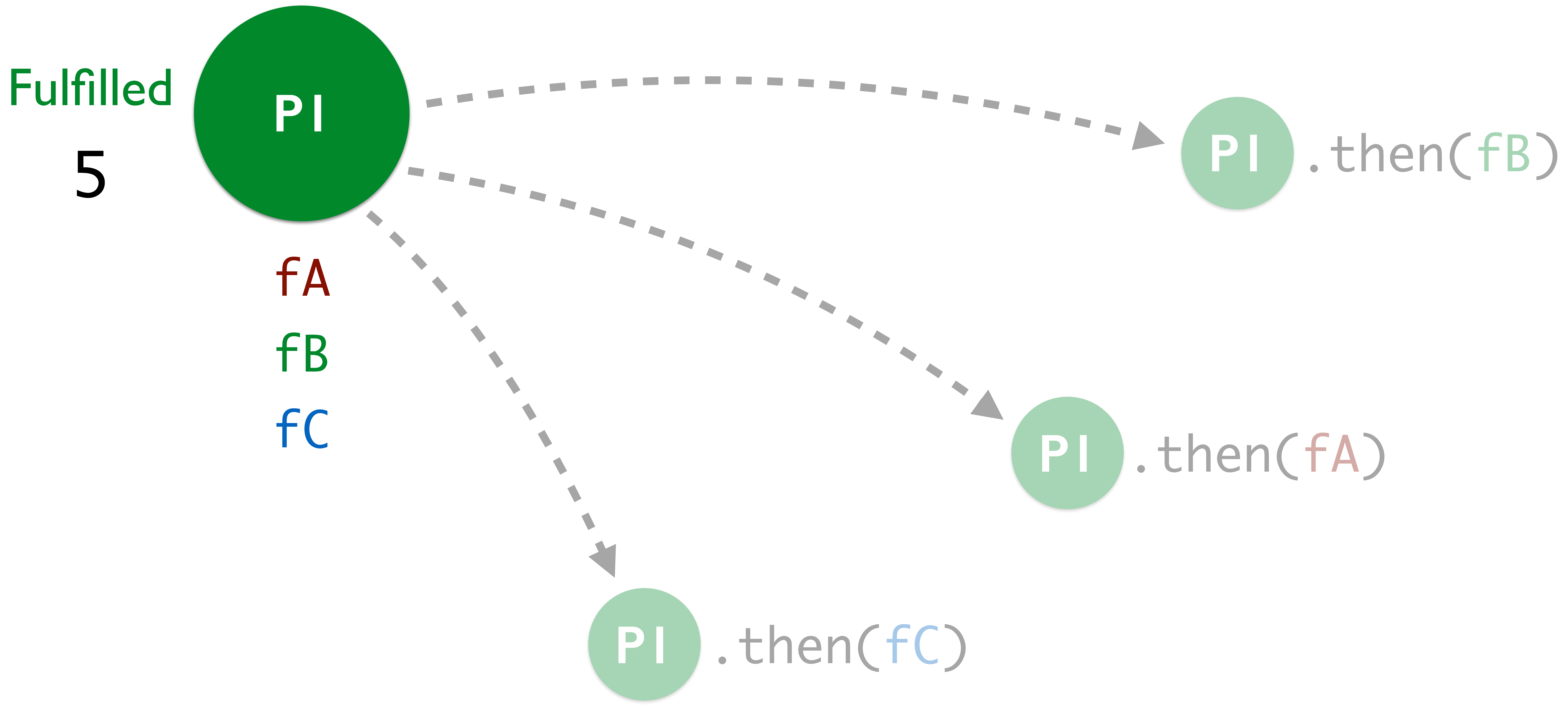
Timing-ambivalent

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

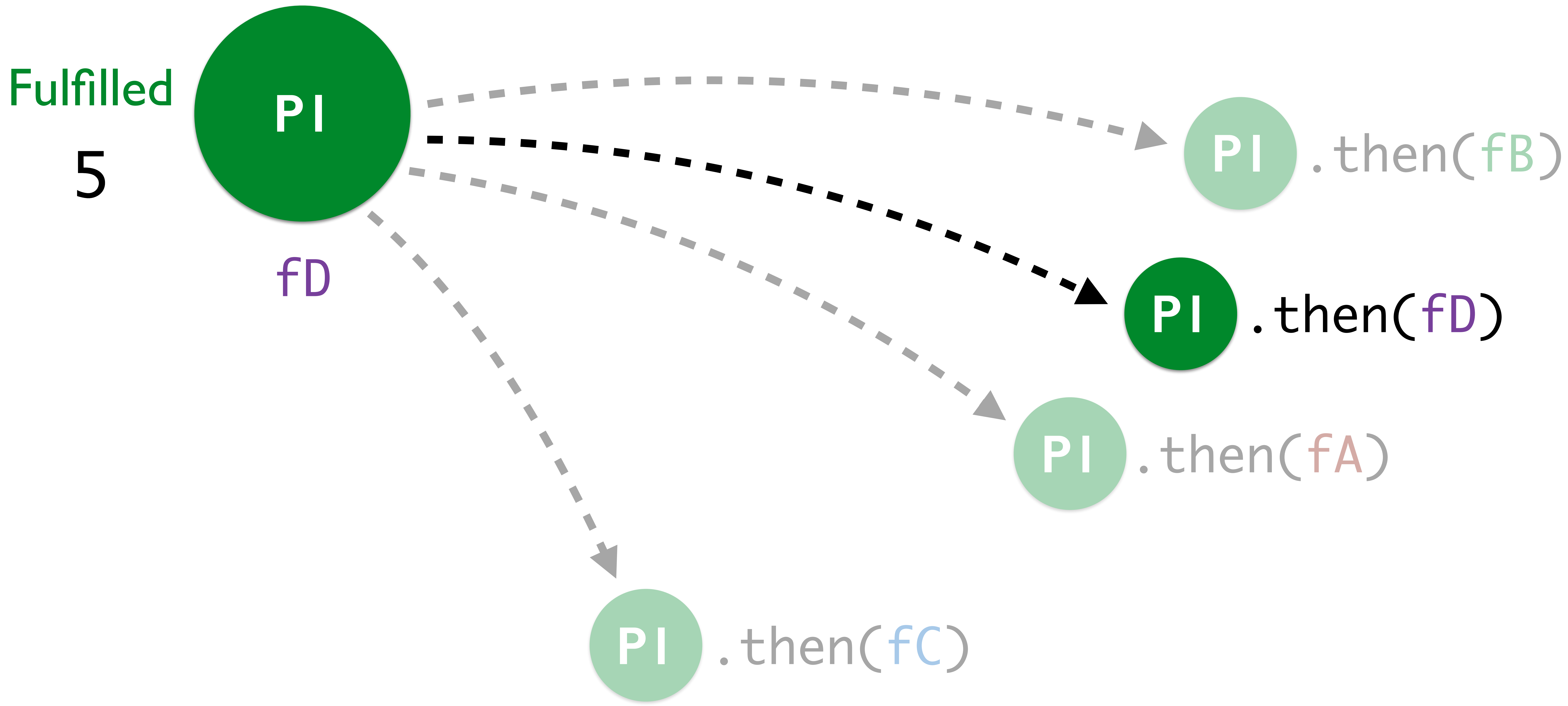
.then on same promise (not chaining!)



.then on same promise (not chaining!)



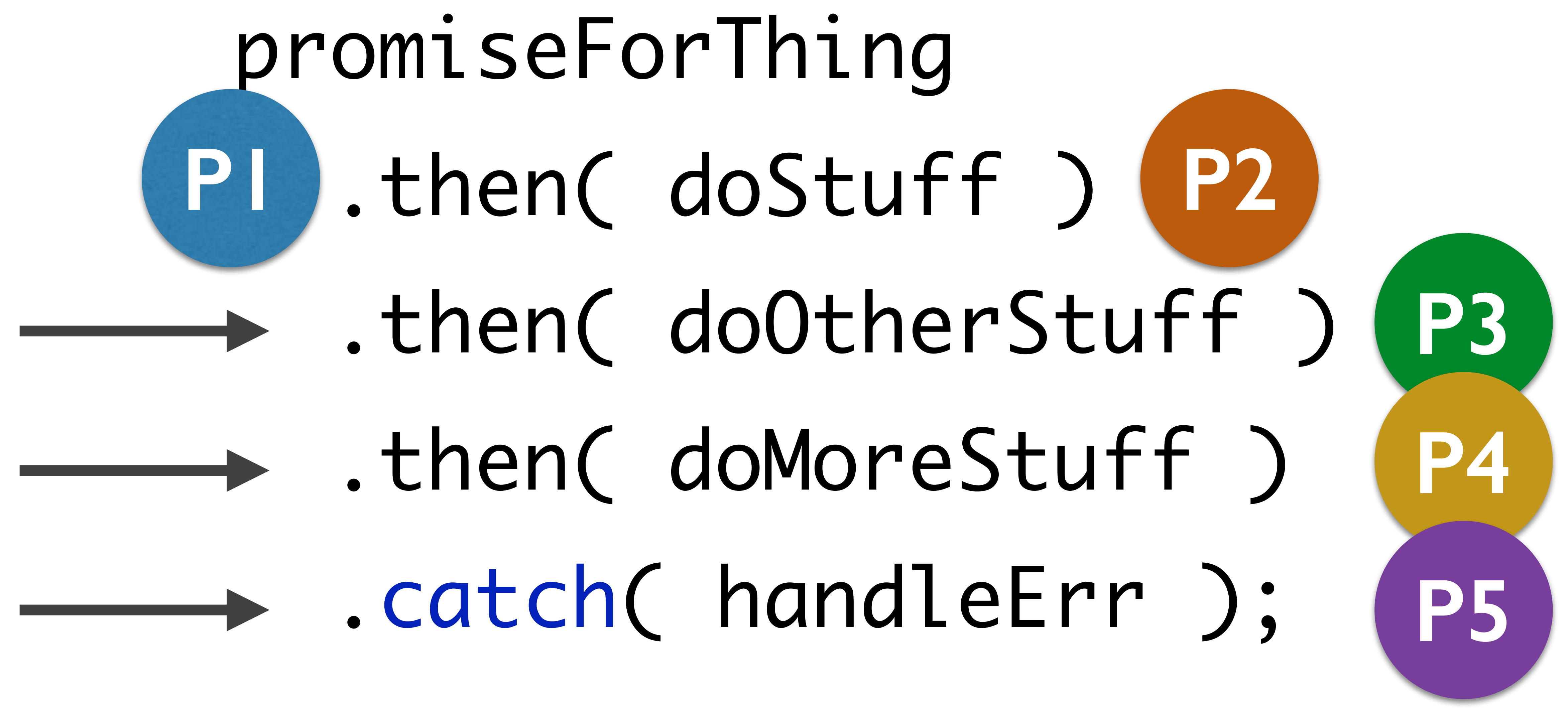
.then on same promise (not chaining!)



the magic: `.then` returns a *new* promise

```
promiseB = promiseA.then(success, fail);
```

This is why we can chain .then



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

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

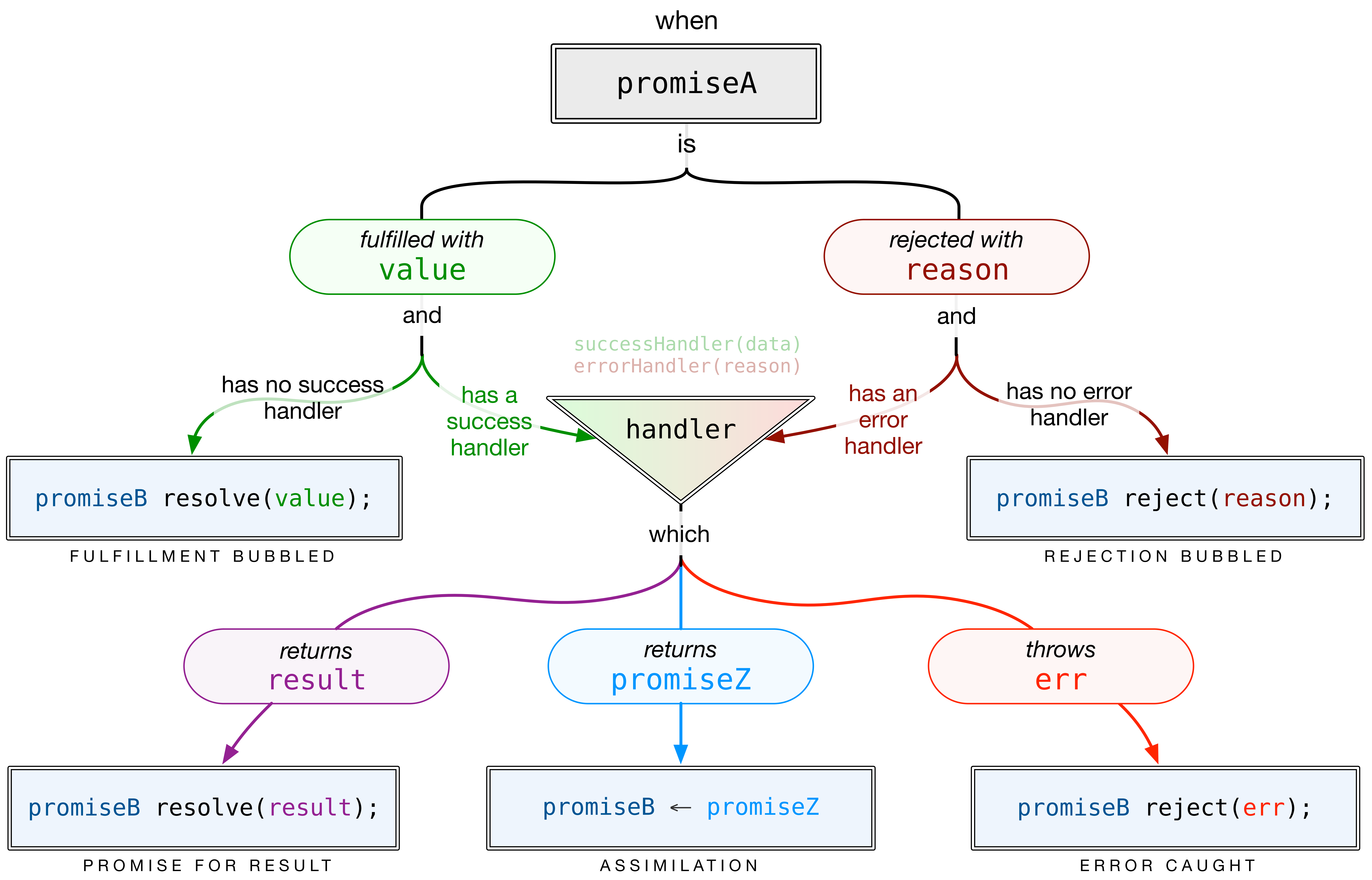
And why we can return from a handler

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

What is promiseB a promise for?

Brace yourselves...

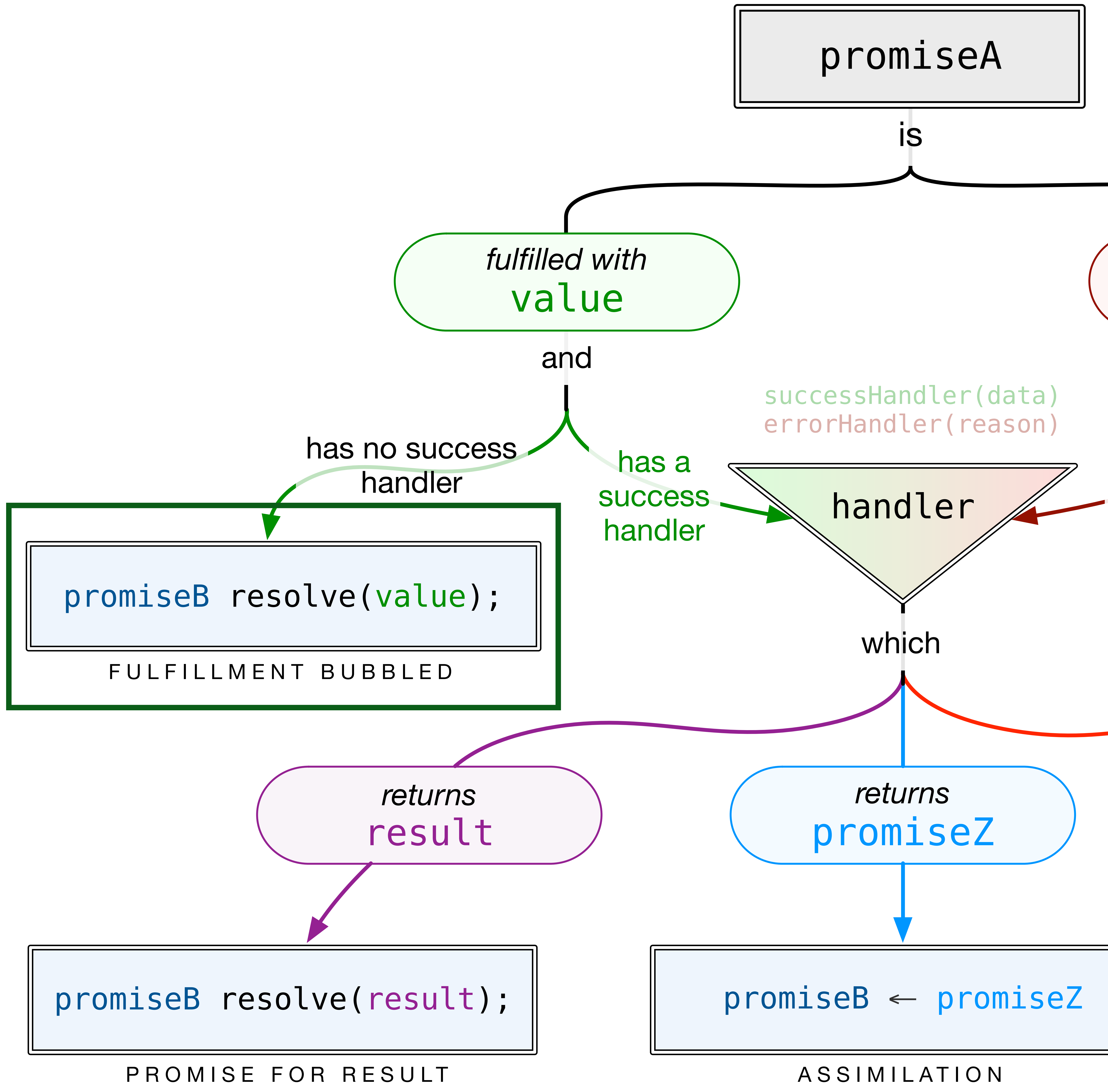
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



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

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

Fulfillment bubbled down to first available success handler:

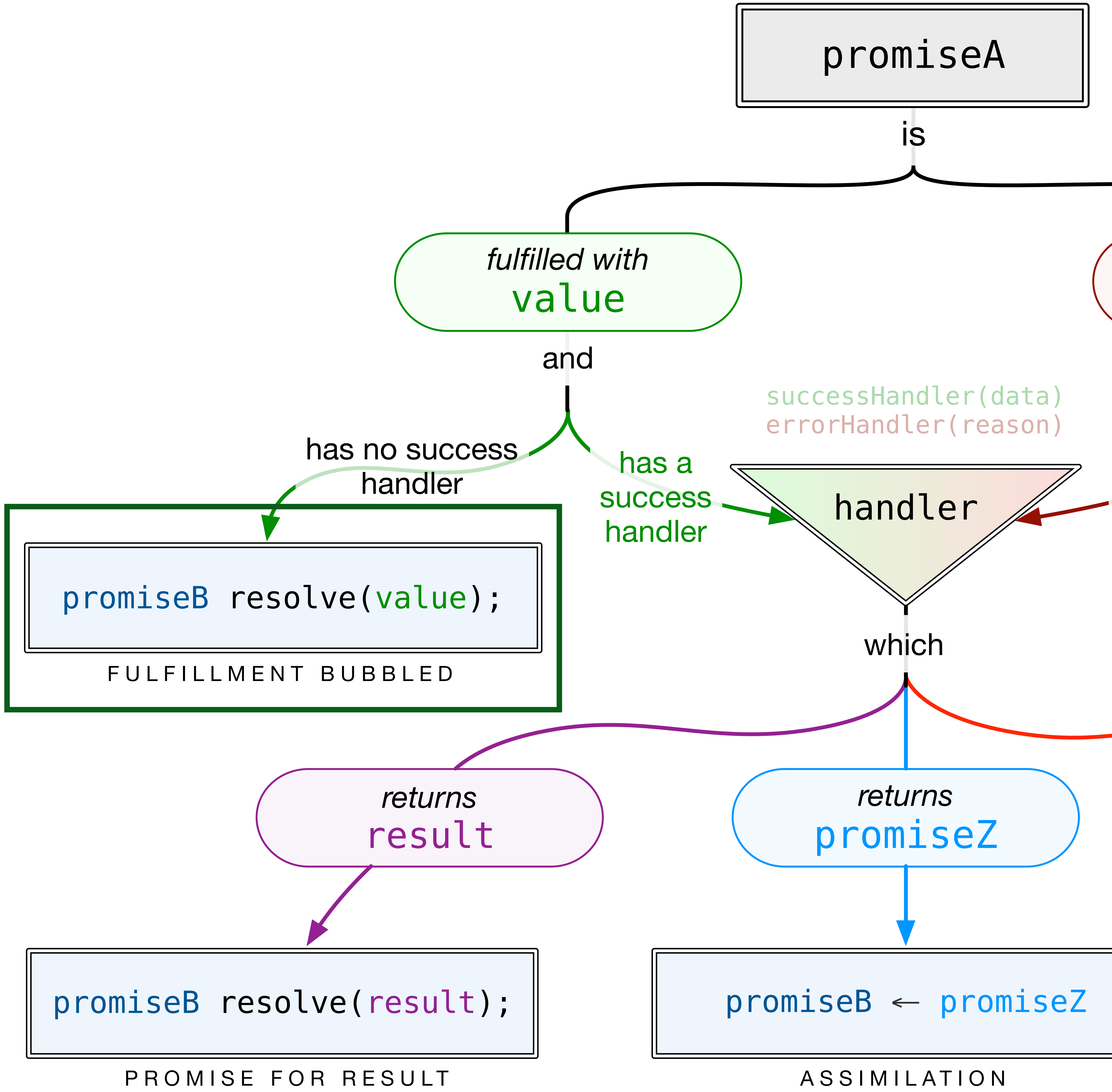
Console log reads "Hello."

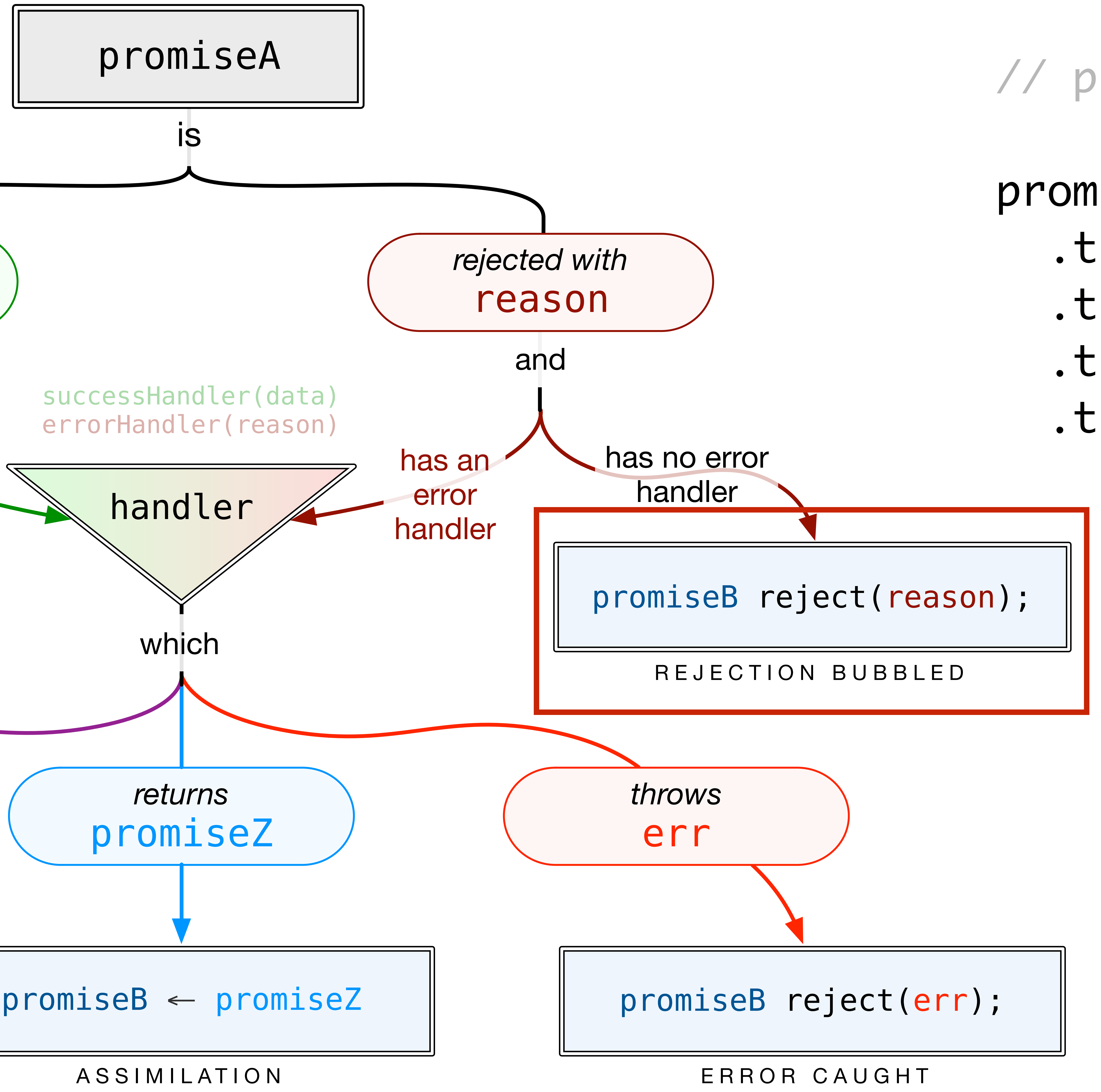


```
// promise0 fulfills with 'Hello.'

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.”

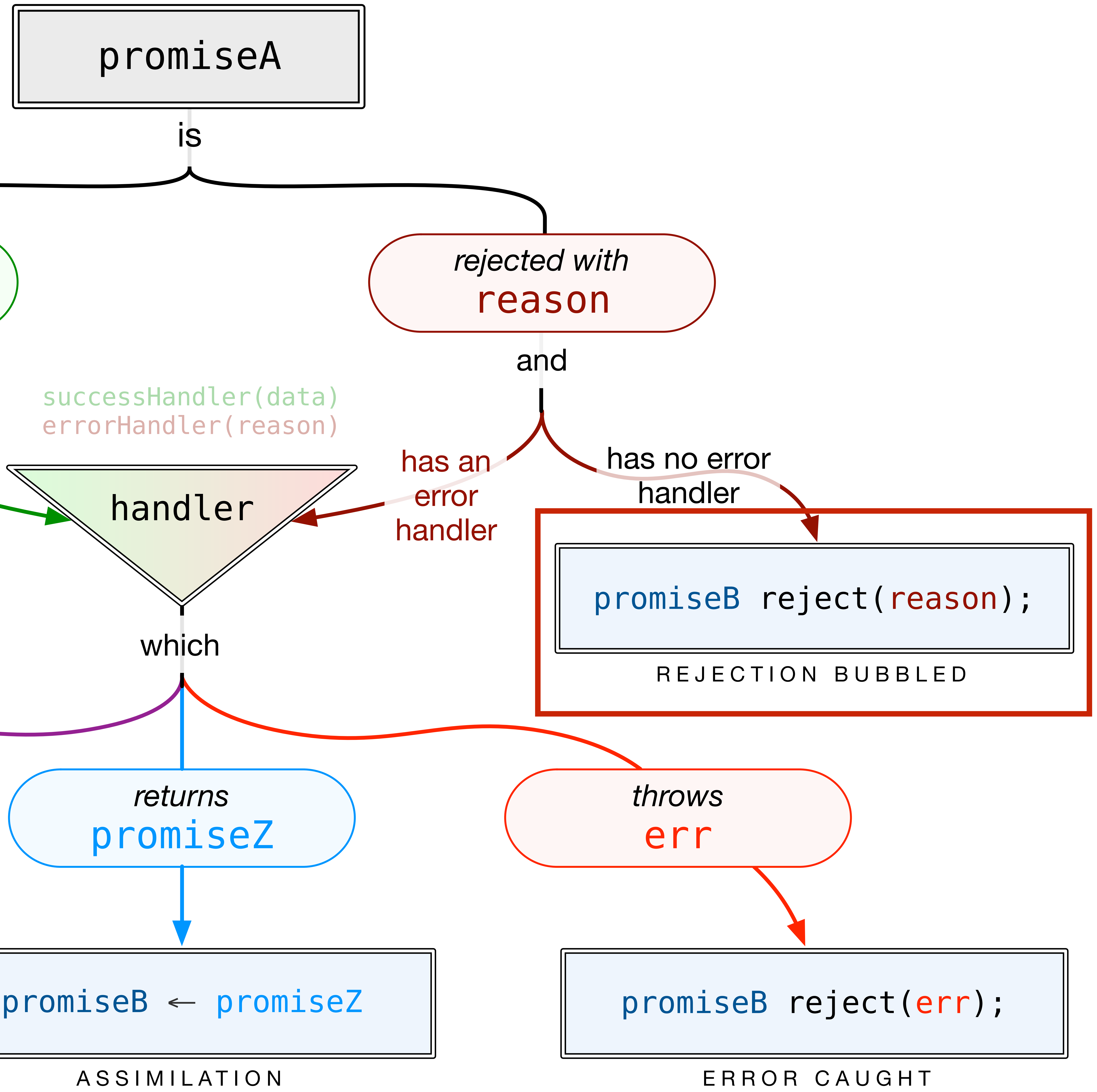




```
// promise0 rejected with 'Sorry'

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
  .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.

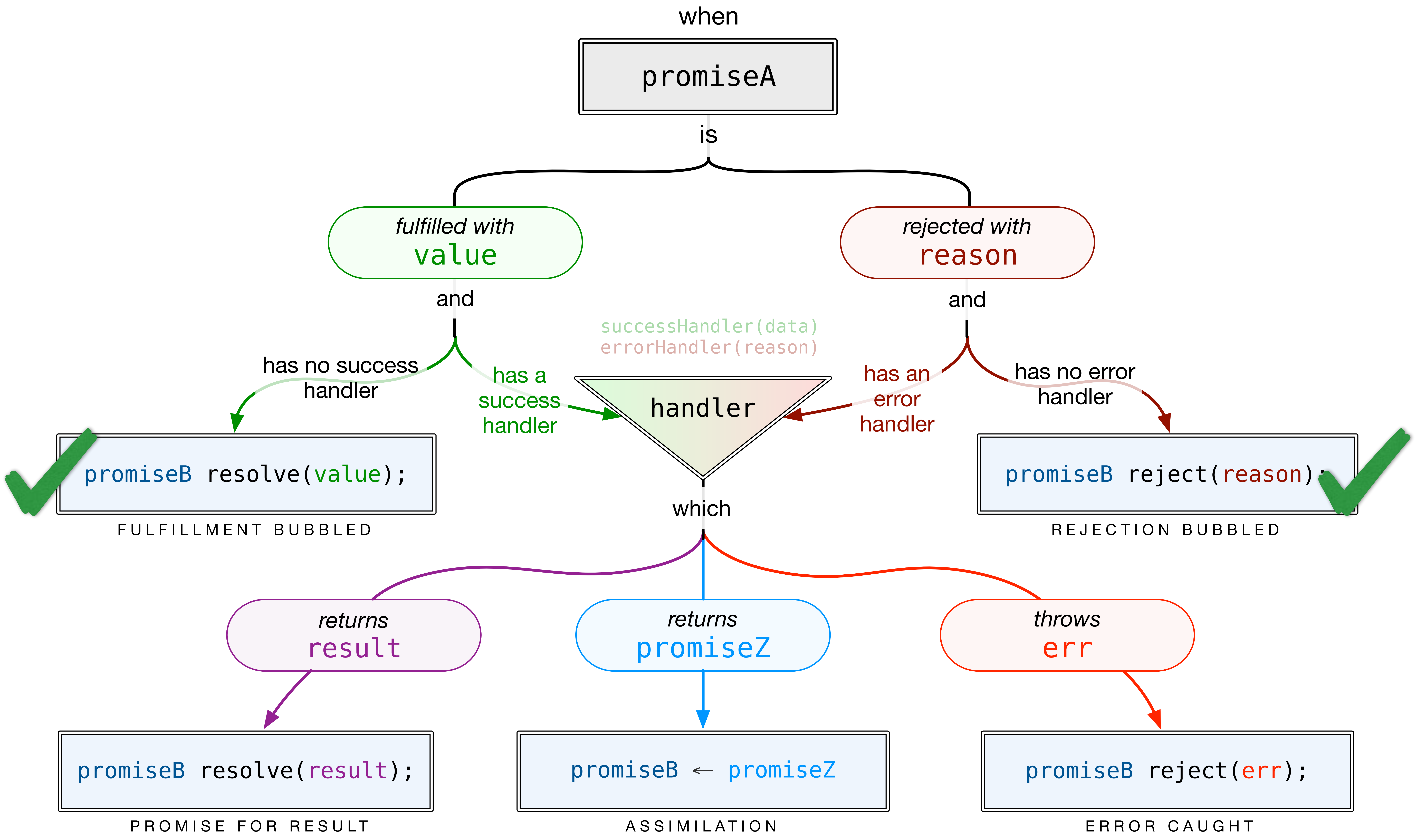
Console log is "Sorry!"

Review: Success & Error Bubbling

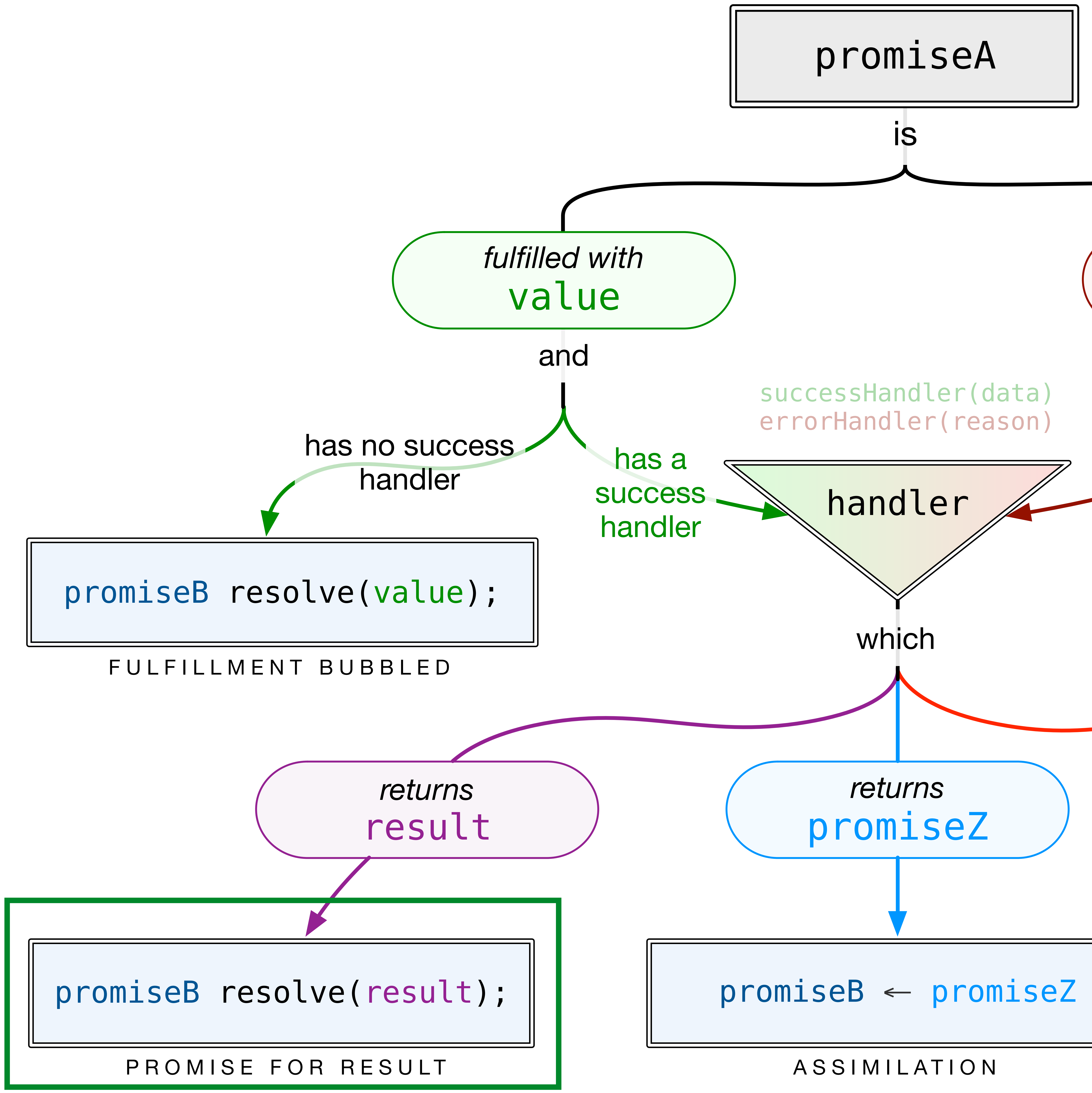
```
// promiseA is fulfilled with 'hello'

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
```

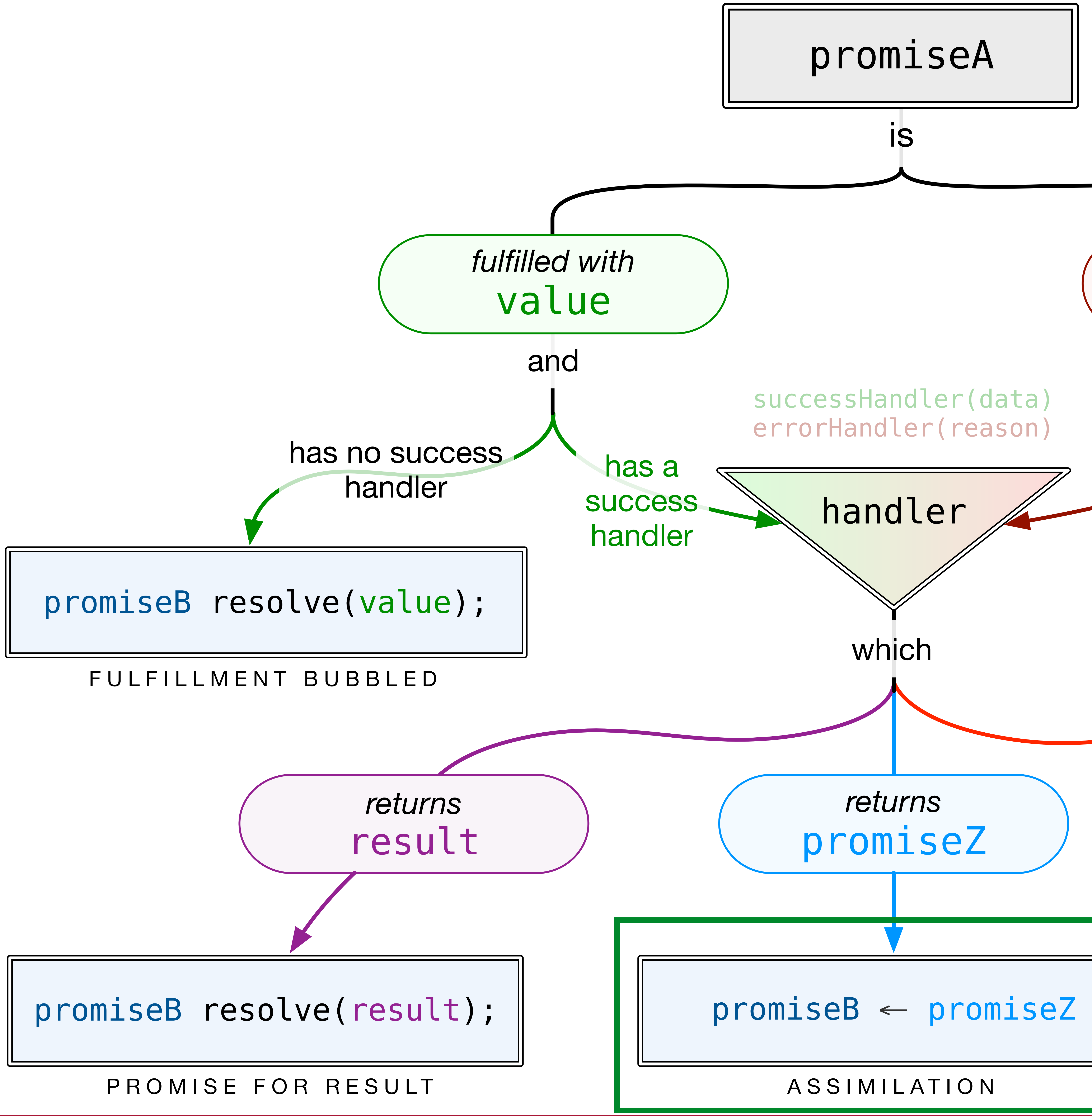
```
promiseB = promiseA.then( [successHandler], [errorHandler] );
```



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



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

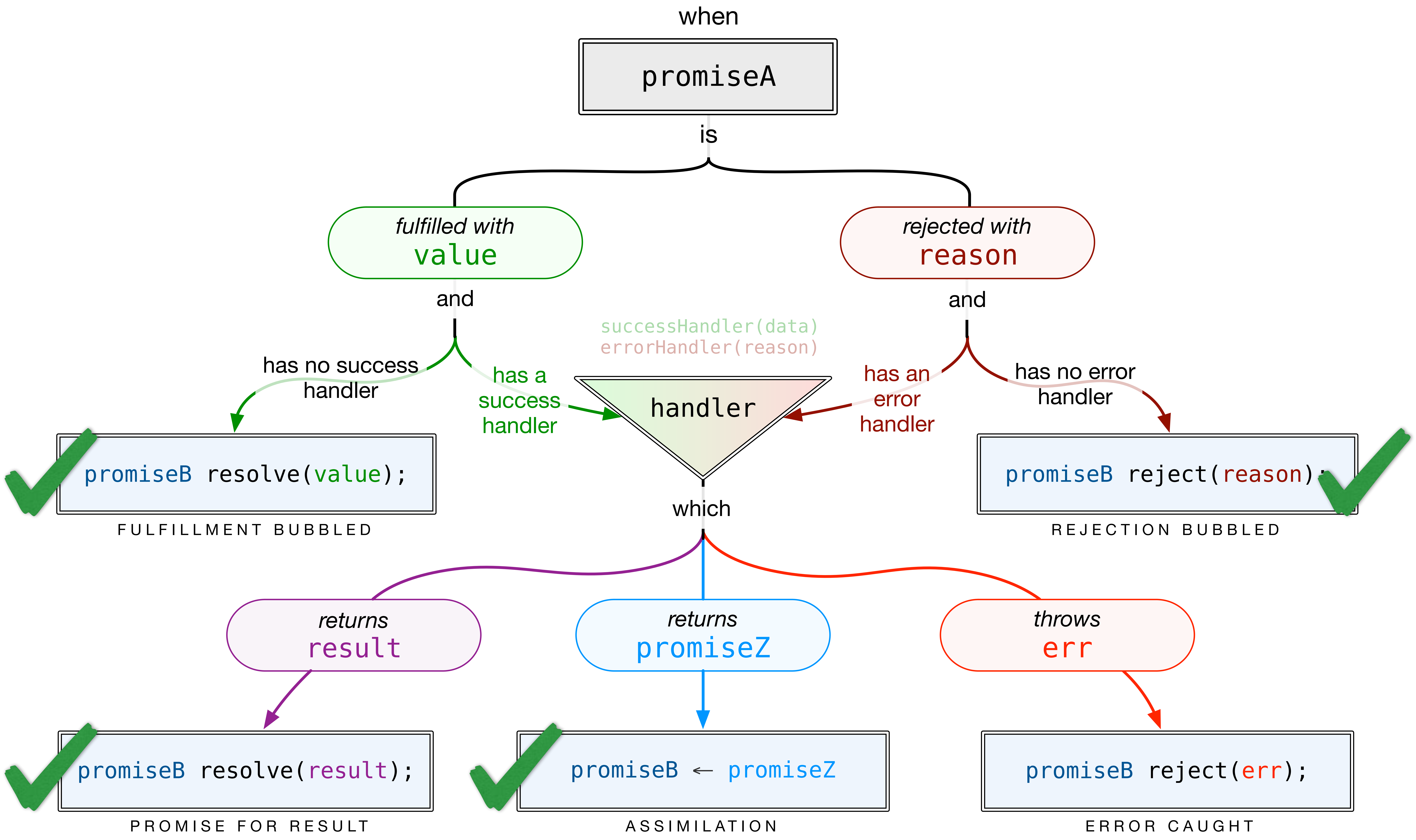


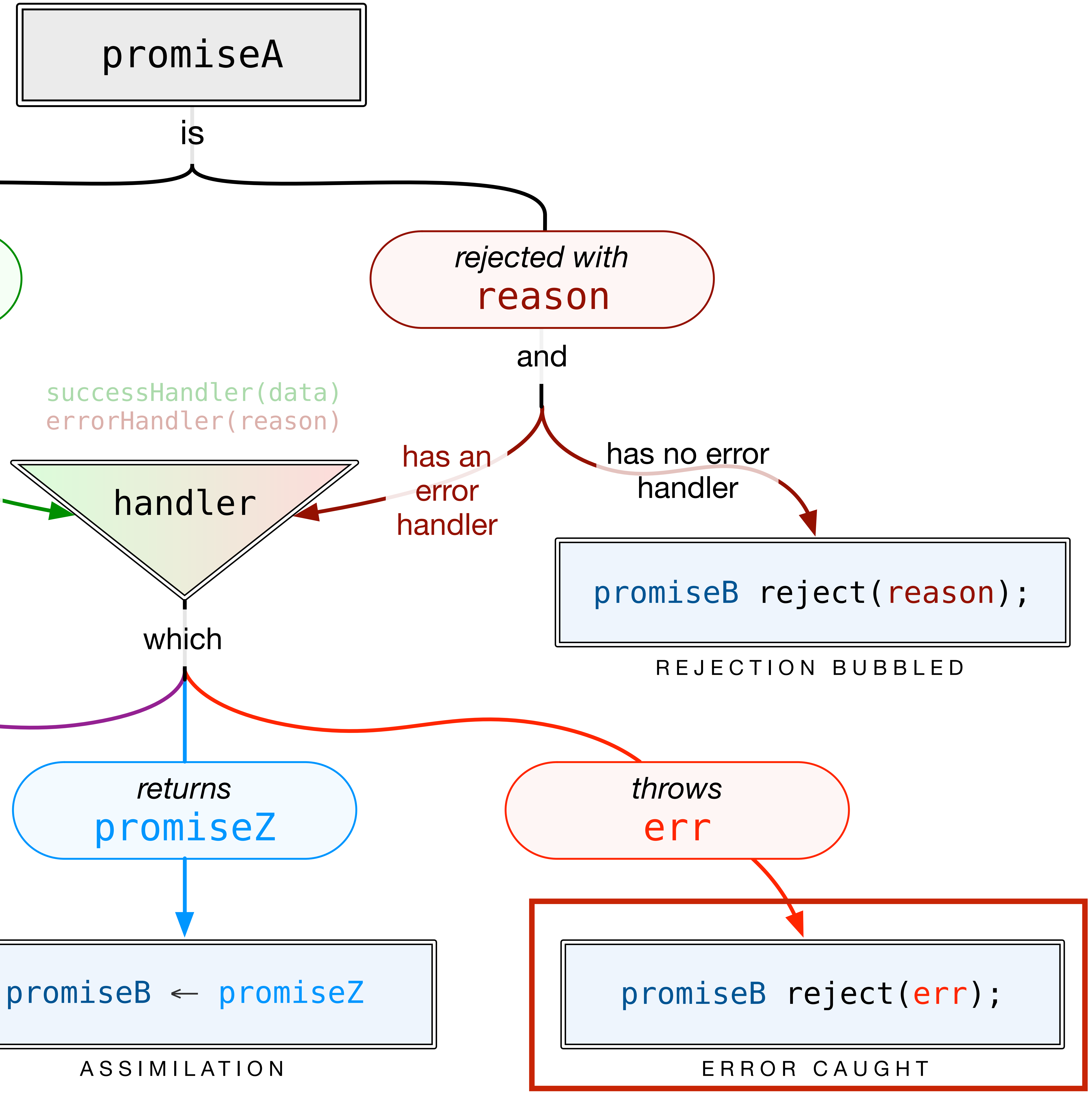
Review: Returning from Handler

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

.then (also .catch) always returns a new promise, so it never throws an error.

Instead, it rejects the outgoing promise

External Resources for Further Reading

- Kris Kowal & Domenic Denicola: Q (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