Promises

based on MDN web docs (Using Promises) and JavaScript.info (Promise)

by Dr. K

Promise

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

— MDN web docs (Using Promises)

Promise: executor

A Promise has a function called an *executor* that runs automatically as soon as the promise is created.

When the executor obtains a result, it calls one of two (JavaScript supplied) callback functions:

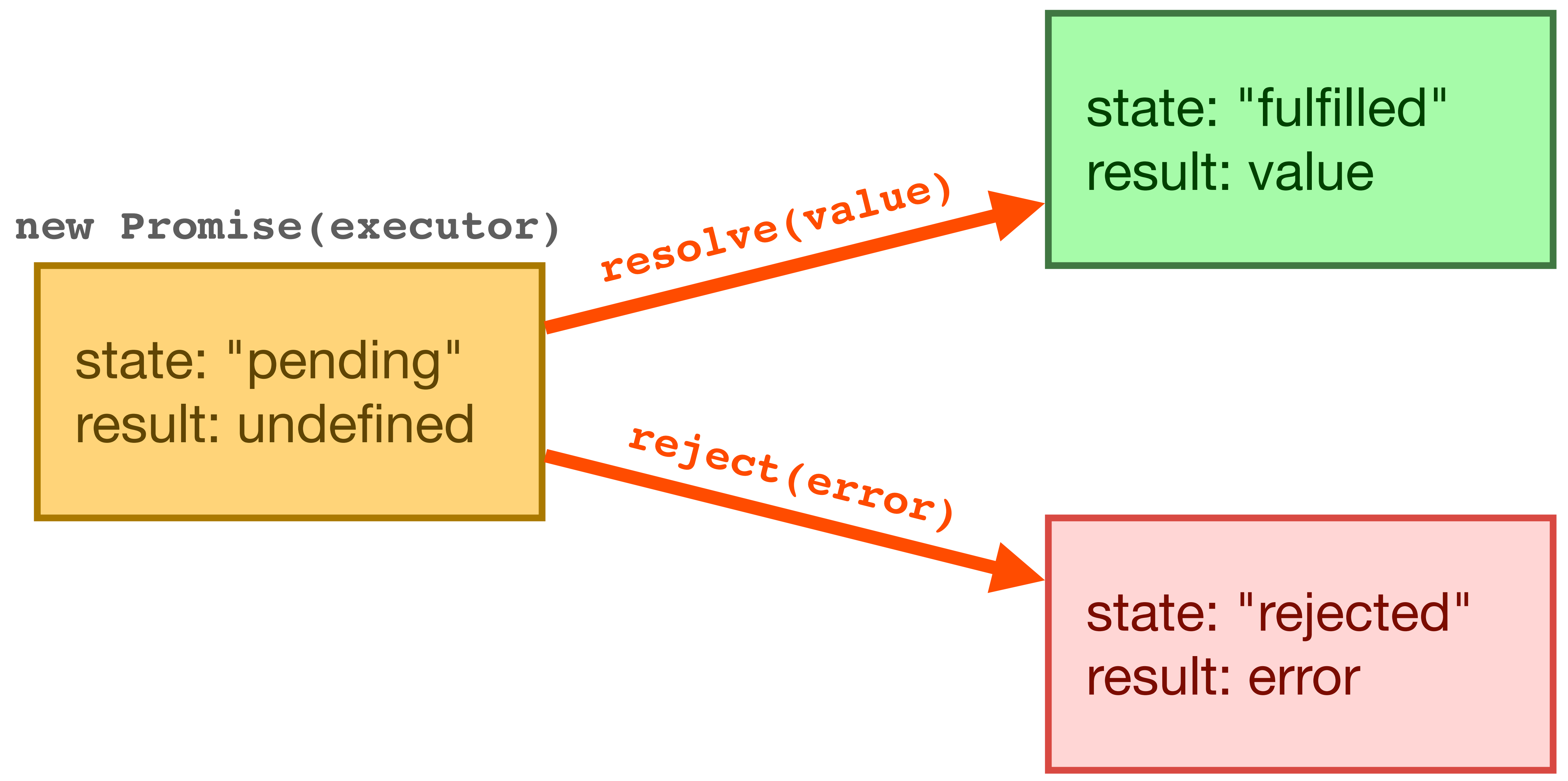
- `resolve(value)` – the job finished successfully with result `value`
- `reject(error)` – an `error` occurred

Promise: internal properties

A Promise object has two internal properties:

- `state` – `"pending"`, `"fulfilled"`, or `"rejected"`
- `result` – `undefined`, `value`, or `error`

Promise



— JavaScript.info (Promise)

Promise consumers: then, catch, finally

Properties **`state`** and **`result`** of the Promise are internal.
We can't directly access them.

But we can use methods **`.then`**, **`.catch`**, and **`.finally`**

Promise: *then* function

We want to invoke code or functions whenever an asynchronous operation is finished. We want some functions to run when the operation succeeds; we want other functions to run when the operation fails. For promises, we can use the **then** function to do both.

```
promise
  .then(
    value => doSomething(value),
    error => handleError(error)
  )
```

the first parameter of the then function handles the value of a fulfilled promise

the second parameter handles the error of a rejected promise

Promise: *then* function

We want to invoke code or functions whenever an asynchronous operation is finished. We want some functions to run when the operation succeeds; we want other functions to run when the operation fails. For promises, we can use the **then** function to do both.

```
promise
   .then(
     value => doSomething(value)
   )
```

if there is no second parameter, nothing is done with an error

if promises are chained, the error continues up the chain

Promise: *catch* function

If we want only to handle an error, we can make the first parameter of a ***then*** function be null, so that nothing is done with the value of a fulfilled promise. Rather than do this, programmers use the ***catch*** function.

This will work but is not used

✗

```
promise
  .then(
    null,
    error => handleError(error)
  )
```

Use this instead

✓

```
promise
  .catch(
    error => handleError(error)
  )
```

these are semantically equivalent

Promise: *finally* function

If we want to clean things up regardless of whether there was an error or not, we can ignore the result and do the same thing for both parameters. Rather than do this, programmers use the ***finally*** function.

Works, but not *exactly* the same

✗

```
promise
   .then(
     () => doCleanup(),
     () => doCleanup()
   )
```

Use this instead

✓

```
promise
   .finally(
     () => doCleanup()
   )
```

these are roughly equivalent

— JavaScript.info (Promise)

Promise chaining

Promise chaining occurs when we have multiple asynchronous tasks that we want to process in sequence. Use a series of ***then*** functions that return a result. The result becomes the value of the next Promise.

this function returns a Promise →

```
doSomething()
  .then(function(result) {
    return doSomethingElse(result);
  })
  .then(function(newResult) {
    return doThirdThing(newResult);
  })
  .then(function(finalResult) {
    console.log('Got the final result: ' + finalResult);
  })
  .catch(failureCallback);
```

— MDN web docs (Using Promises)

Promise chaining

Promise chaining occurs when we have multiple asynchronous tasks that we want to process in sequence. Use a series of ***then*** functions that return a result. The result becomes the value of the next Promise.

same chain but expressed with arrow functions

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

Important: Always return results, otherwise callbacks won't catch the result of a previous promise. With arrow functions, `() => x` is short for `() => { return x; }`

— MDN web docs (Using Promises)