Asynchronous JavaScript

Callbacks

Promises

Async/Await

Synchronous vs. Asynchronous

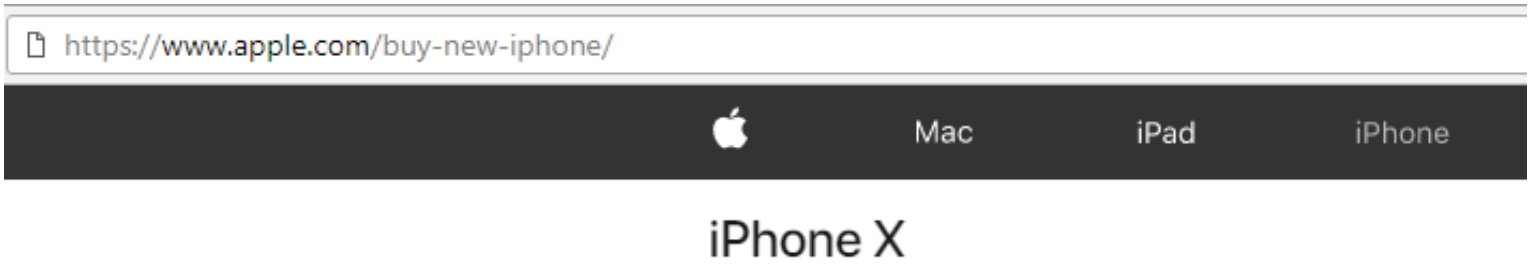
Buying newly released iPhone

- **Synchronous**:
  - You go to an Apple store
  - Wait impatiently in a queue, then pay for the phone and take it home

Synchronous vs. Asynchronous

Buying newly released iPhone

- **Asynchronous**:
  - You order the phone online from apple.com,
  - Then get on with other things in your life.
  - At some point in the future, the phone will be shipped. The postman will raise a knocking event on your door so that the phone can be delivered to you.



Sync Programming is Easy

```
function getStockPrice(name) {
    let symbol = getStockSymbol(name);
    let price = getStockPrice(symbol);
    return price;
}
```

Call a function,
suspend the caller
and wait for the return value to arrive

Synchronous Programming Problems

- CPU demanding tasks delay execution of all other tasks => **UI may become unresponsive**
- Accessing resources such as files blocks the entire program
  - Especially problematic with web resources
    - Resource may be large
    - Server may hang
    - Slow connection means slow loading causing UI blocks

Why use Async Programming?

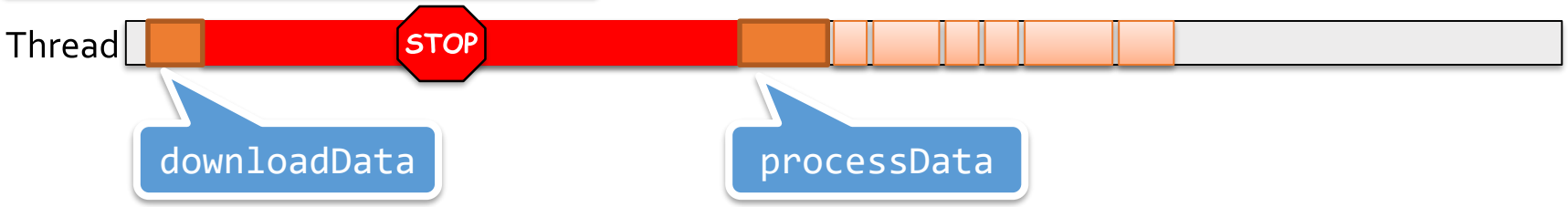
- JavaScript is single-threaded
  - Long-running operations block other operations
- Async Programming is required to **prevent blocking** on long-running operations
- Benefits:
  - ***Responsiveness**: prevent blocking of the UI*

    => Doesn't lock UI on long-running computations
  - Better server-side ***Scalability**: prevent blocking of request-handling threads*

Synchronous vs. Asynchronous

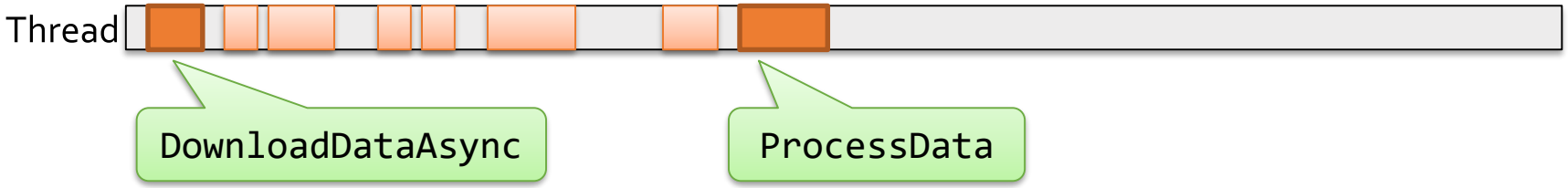
```
let data =
downloadData(...);
processData(data);
```

Synchronous → Wait for result before returning



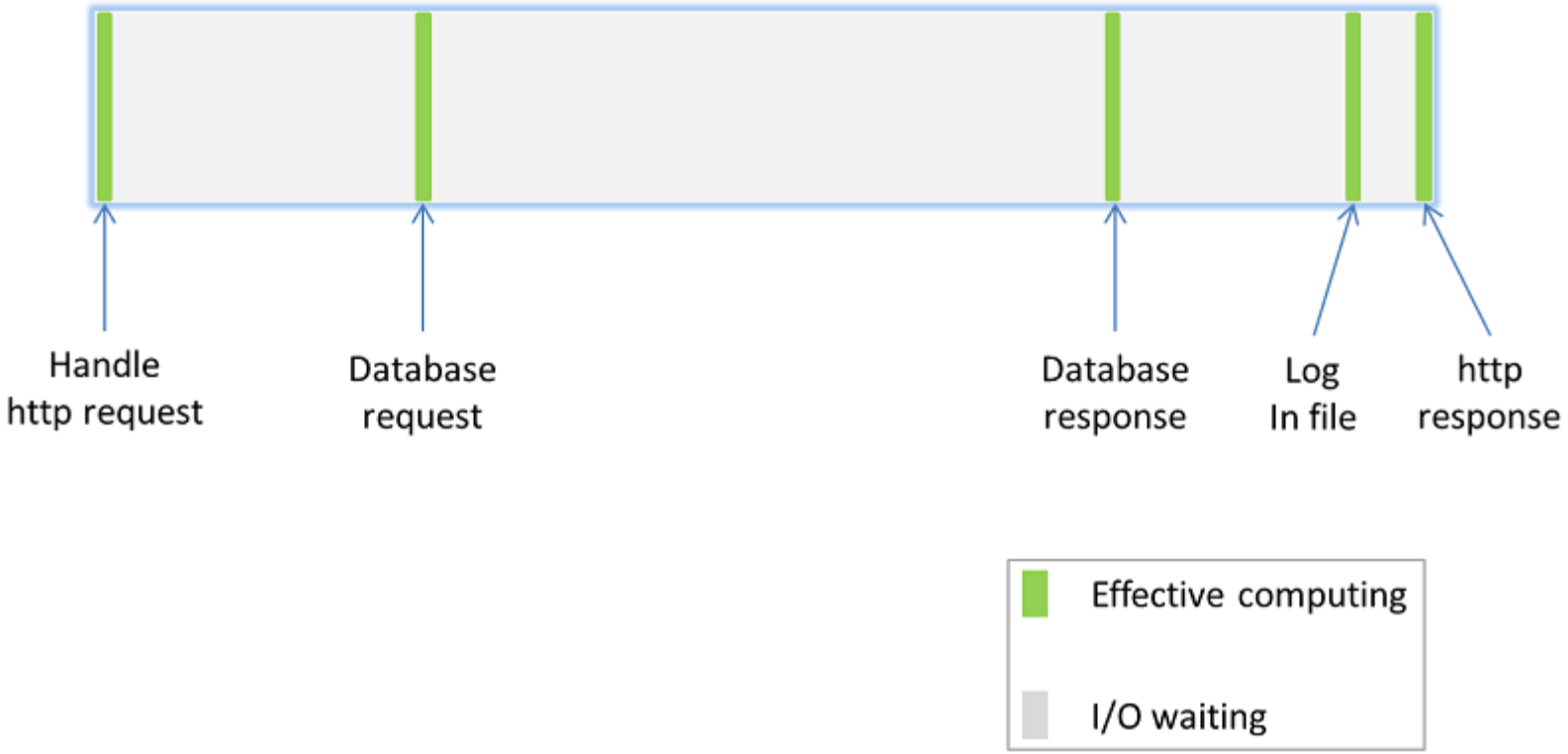

```
DownloadDataAsync(... , data => {
    ProcessData(data);
});
```

Asynchronous → Return now, call back with result



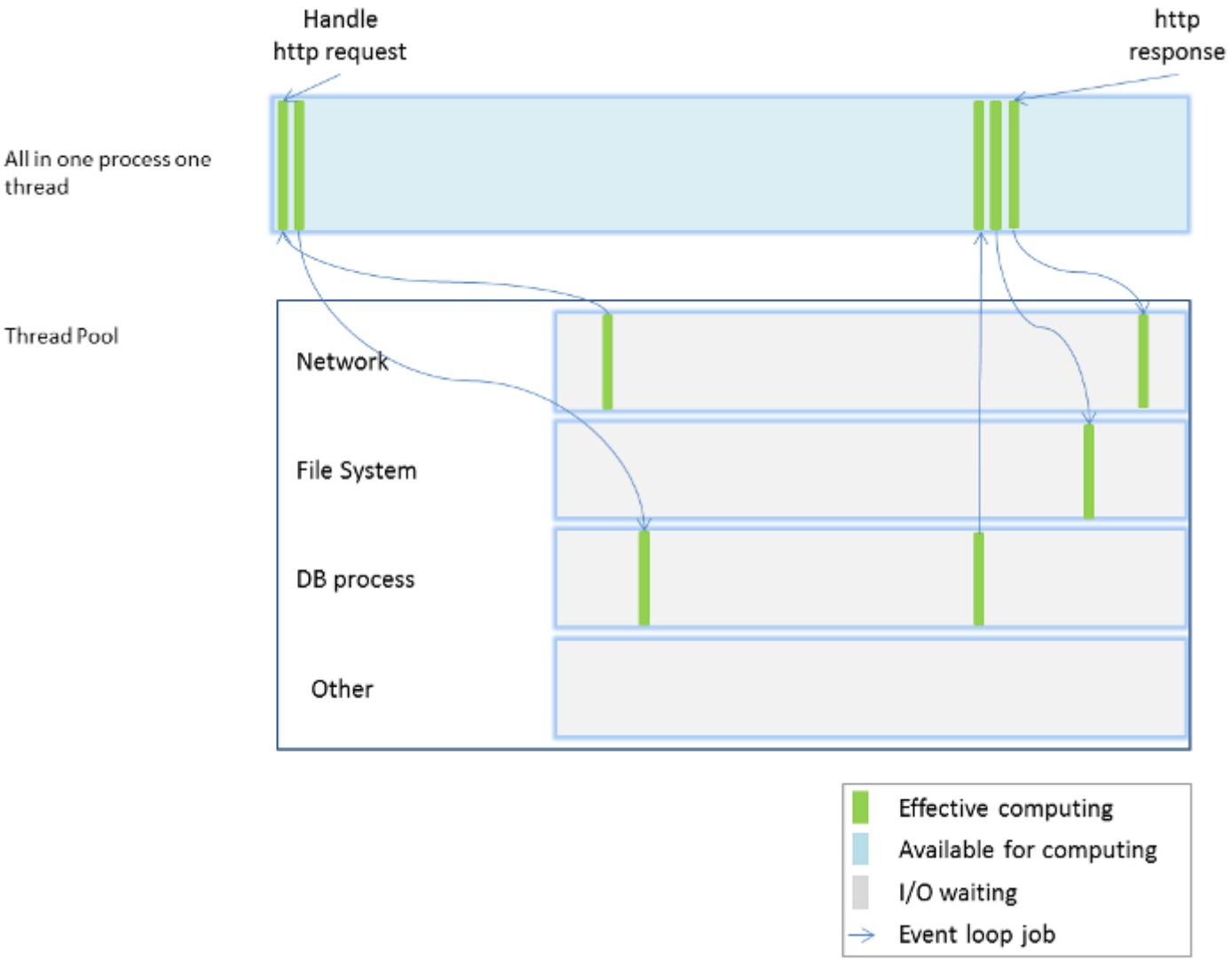
Processing allowed **before** current **execution** is **done**

I/O Latency



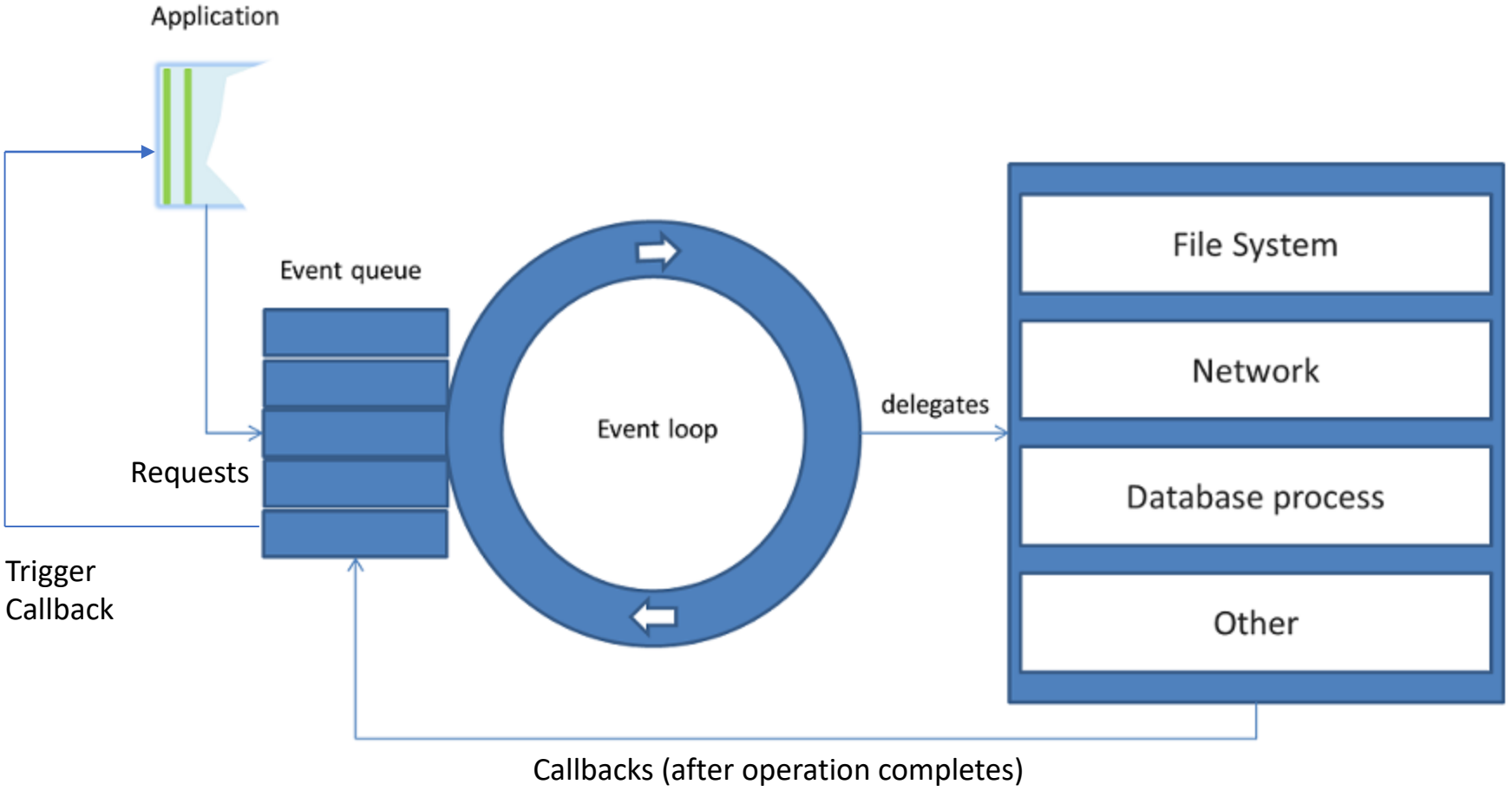
Most of the time the process waits for I/O (disk, network)

Reduce Latency with Event Loop



Delegate the I/O part to a Thread Pool

Event Loop



Delegate I/O tasks and manage callbacks

Watch https://www.youtube.com/watch?v=8aGhZQkoFbQ

Asynchronous programming techniques

Async JavaScript programming can be done using either:

- Callbacks
- Promises
- Async/Await

Callback-oriented Programming

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function
  - The outer function can pass arguments
- Examples of callbacks:
  - E.g., **`navigator.geolocation.getCurrentPosition`** takes a callback argument
- Problems:
  - Heavily nested functions are hard to understand => **Callback hell** i.e., non-trivial to follow path of execution
  - Errors and exceptions are a hard to handle

Callback Example

```
function getLocation() {
    navigator.geolocation.getCurrentPosition(showPosition);
}

function showPosition(position) {
    let p = document. querySelector("#demo");
    p.innerHTML += `Latitude: ${position.coords.latitude}
       <br>Longitude: ${position.coords.longitude} <BR>`;
}
```

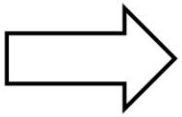
Callback Hell…

```
function getStockPrice(name, cb) {
    getStockSymbol(name, (error, symbol) => {
        if (error) {
            cb(error);
        }
        else {
            getStockPrice(symbol, (error, price) => {
                if (error) {
                    cb(error);
                }
                else {
                    cb(price);
                }
            })
        }
    })
}
```

Promises solves the Callback Hell…

CALLBACK

```
getData(a ⇒ {
    getMoreData(a, b ⇒ {
        getMoreData(b, c ⇒ {
            getMoreData(c, d ⇒ {
                getMoreData(d, e ⇒ {
                    console.log(e);
                });
            });
        });
    });
});
```



PROMISES

```
getData()
.then(a ⇒ getMoreData(a))
.then(b ⇒ getMoreData(b))
.then(c ⇒ getMoreData(c))
.then(d ⇒ getMoreData(d))
.then(e ⇒ console.log(e));
```

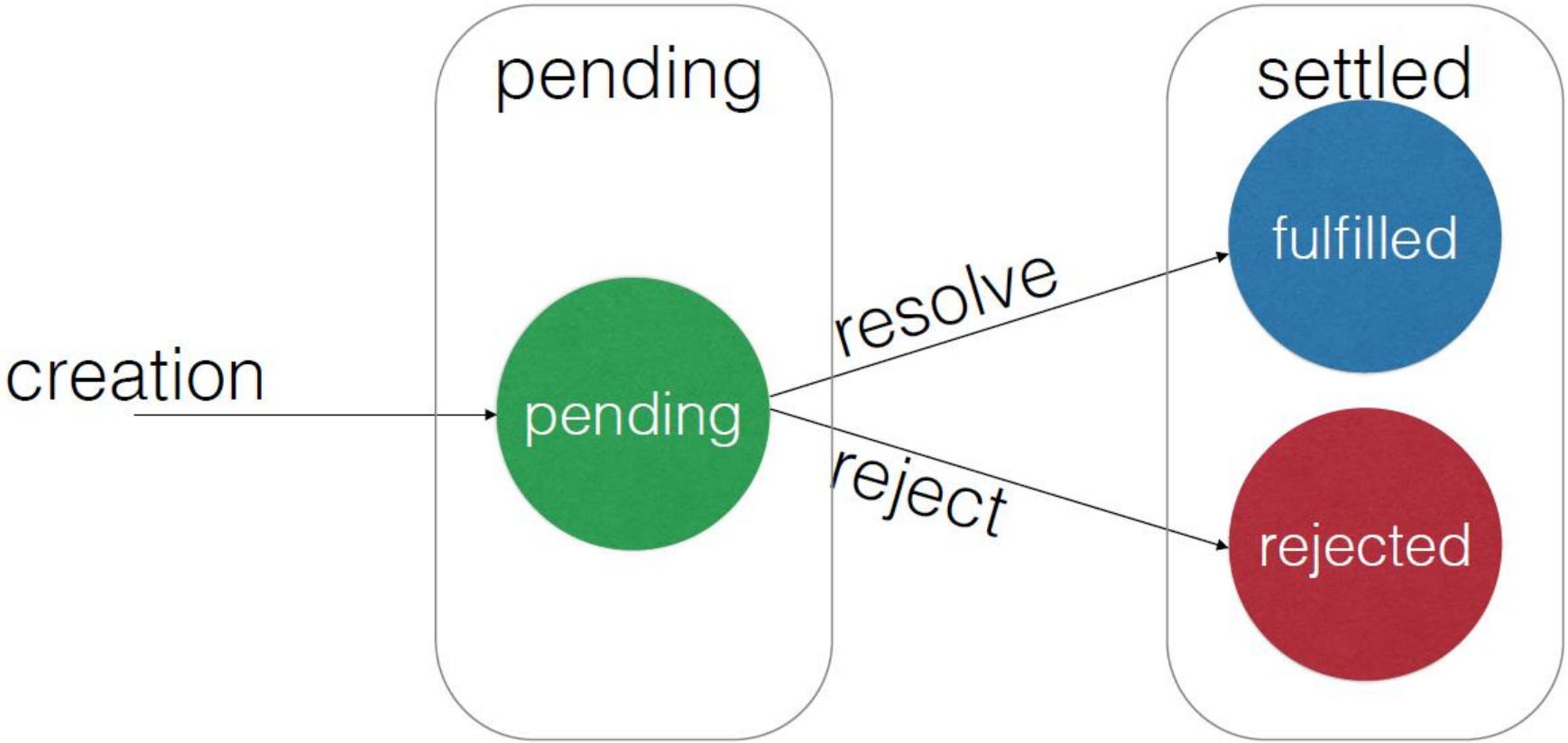
Promises

- Promise = object that represents an eventual (future) value
- A producer returns a promise which it can later fulfill or reject
- Promise has one of three states: pending, fulfilled, or rejected
- Consumers listen for state changes with **.then** method:

```
promise.then(onFulfilled)
       .catch(onRejected)
       .finally(() => console.log('done!'));
```

- onFulfilled is function to process the received results
- onRejected is a function to handle errors

State of a Promise



How to create a Promise

```
let promise = new Promise((resolve, reject)
=> {
    try {
        …
        resolve(value);
    } catch(e)  {
        reject(e);
     }
});
```

Example: Writing a Promise

- Wrapping **fs.readFile** in a promise

```
function getStudent(studentId) {
    return new Promise( (resolve, reject) => {
        fs.readFile('data/student.json', function (err, data) {
                if (err) {
                    reject(err);
                } else {
                    const students = JSON.parse(data);
                    const student = students.find(s => s.studentId === studentId);
                    resolve(student);
                }
            }
        );
    });
}
```

Example - Getting a resource from Url using node-fetch API

- Fetch content from the server

```
let url = "https://api.github.com/users/github";
fetch(url).then(response => response.json())
          .then(user => {
              console.log(user);
           })
          .catch(err => console.log(err));
```

- Fetch returns a Promise. Promise-fulfilled event (**.then**) receives a response object.
- **.json()** method is used to get the response body into a JSON object

sync vs. async

- **sync**

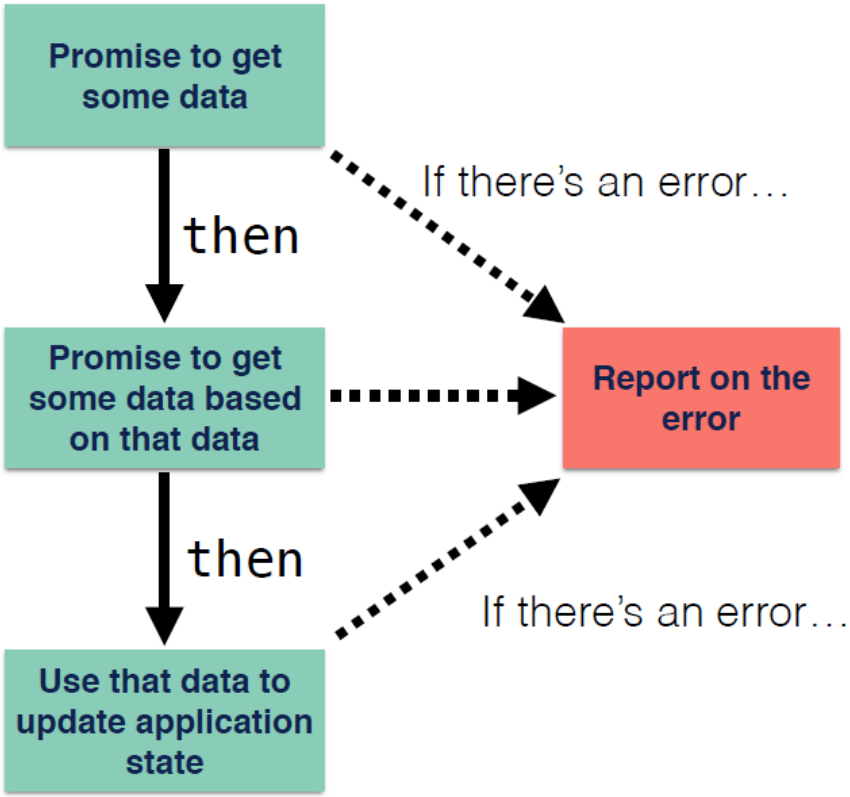
```
function getStockPrice(name) {
    let symbol = getStockSymbol(name);
    let price = getStockPrice(symbol);
    return price;
}
```

- **async**

```
function getStockPrice(name) {
    return getStockSymbol(name).
     then(symbol => getStockPrice(symbol));
}
```

Chaining Promises

Chaining Promises organize many steps that need to happen in order, with each step happening asynchronously



- See example @ http://jsfiddle.net/erradi/cxg5exox/

Chaining Promises

```
getUser()
  .then(function(user) {
    return getRights(user);
  })
  .then(function(rights) {
    updateMenu(rights);
  })
```

Better Syntax

```
getUser()
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

Promise Utilities

- **Promise.all** calls many promises and returns only when all the specified promises have completed or been rejected. The result returned is an array of values returned by the completed promises.

```
Promise.all([p1, p2, ..., pN]).then(allResults =>
{ ... });
```

- **Promise.race** calls two or more promises and returns the first response received (an ignores the remaining ones)

```
Promise.race([p1, p2, ..., pN]).then(firstResult
=> { ... });
```

async / await

- Allows easier composition of promises compared to chaining using **.then**
- **async** function can halt without blocking and waits for the result of a promise

```
async function getStudentCourses(studentId) {
    let student = await getStudent(studentId);
    student.courses = await getCourses(student.courseIds);
    return student;
}

let studentId = 2015002;
getStudentCourses(studentId)
    .then( student =>  console.log( JSON.stringify(student, null, 2)) )
    .catch( err => console.log(err) );
```