

Asynchronous JavaScript

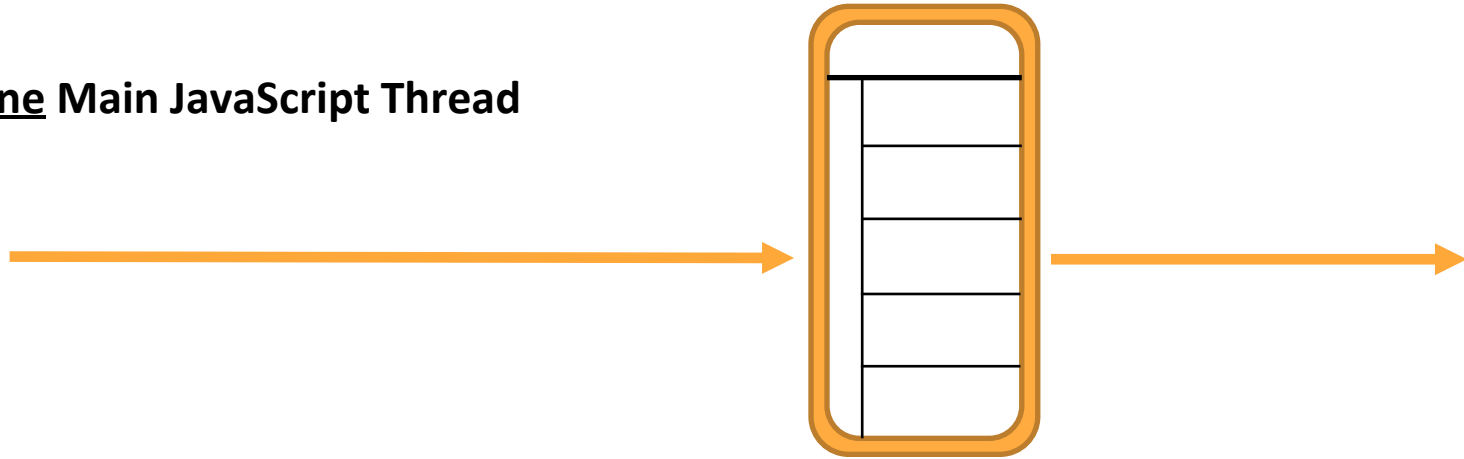
Callbacks

Promises

async/await

Avoid Long Running Tasks on the Main Thread

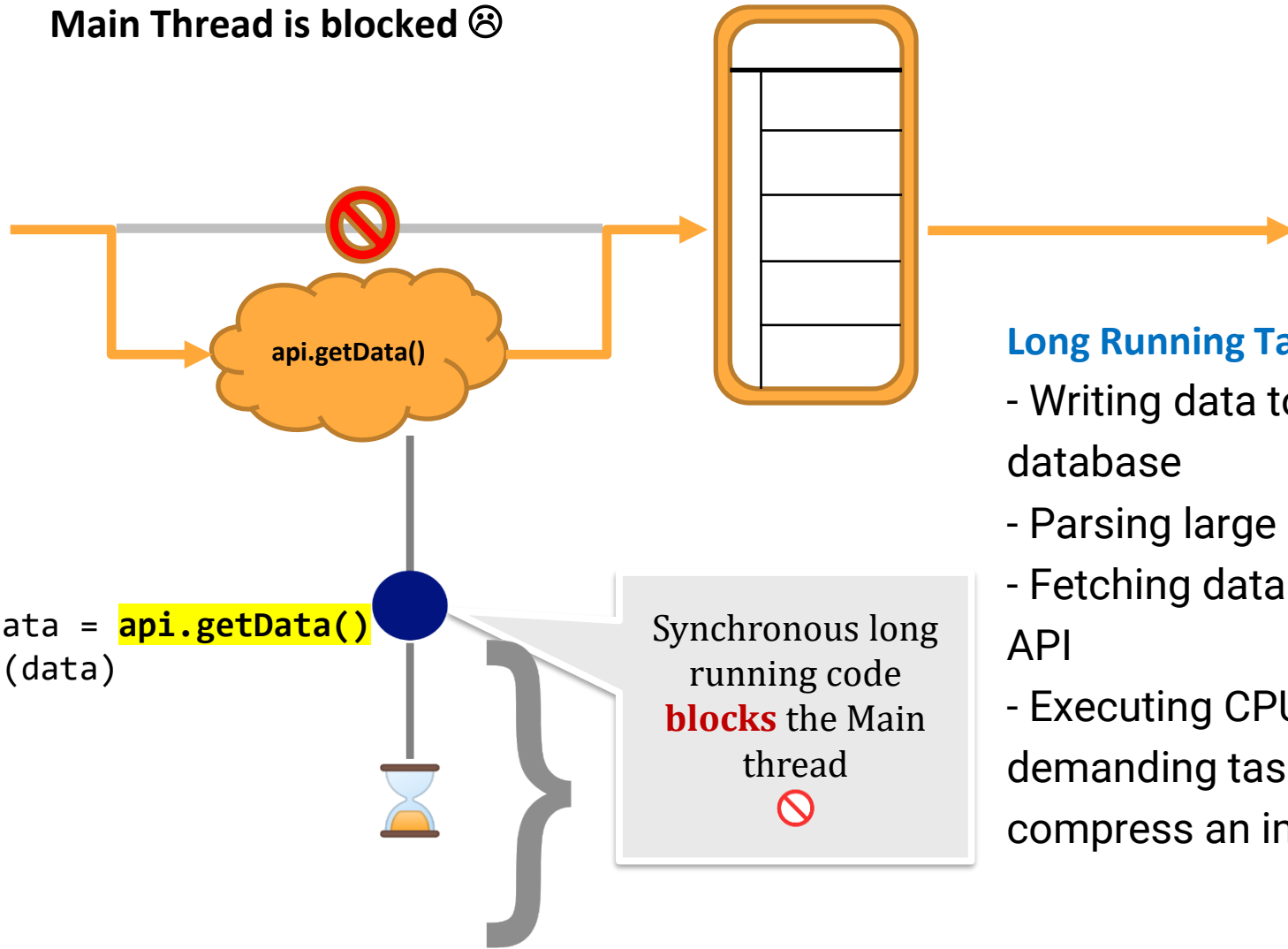
One Main JavaScript Thread



To guarantee a great user experience, it's essential to **avoid blocking the main thread** as it used to handle UI events on client-side (or handles incoming requests on the server-side)

Long Running Task on the Main Thread

Main Thread is blocked ☹️



Long Running Tasks include:

- Writing data to a file / database
- Parsing large JSON file
- Fetching data from Web API
- Executing CPU demanding task (e.g., compress an image)

```
const data = api.getData()  
display(data)
```

Synchronous long
running code
blocks the Main
thread



Need to use **Asynchronous** programming to avoid blocking the main thread

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.



iPhone X

Sync Programming is Easy

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

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

Synchronous Programming Problems

- I/O and CPU demanding tasks delay execution of all other tasks => **UI may become unresponsive**
- Especially problematic with accessing 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*



Asynchronous programming techniques

How to execute a long running tasks without blocking the Main thread?

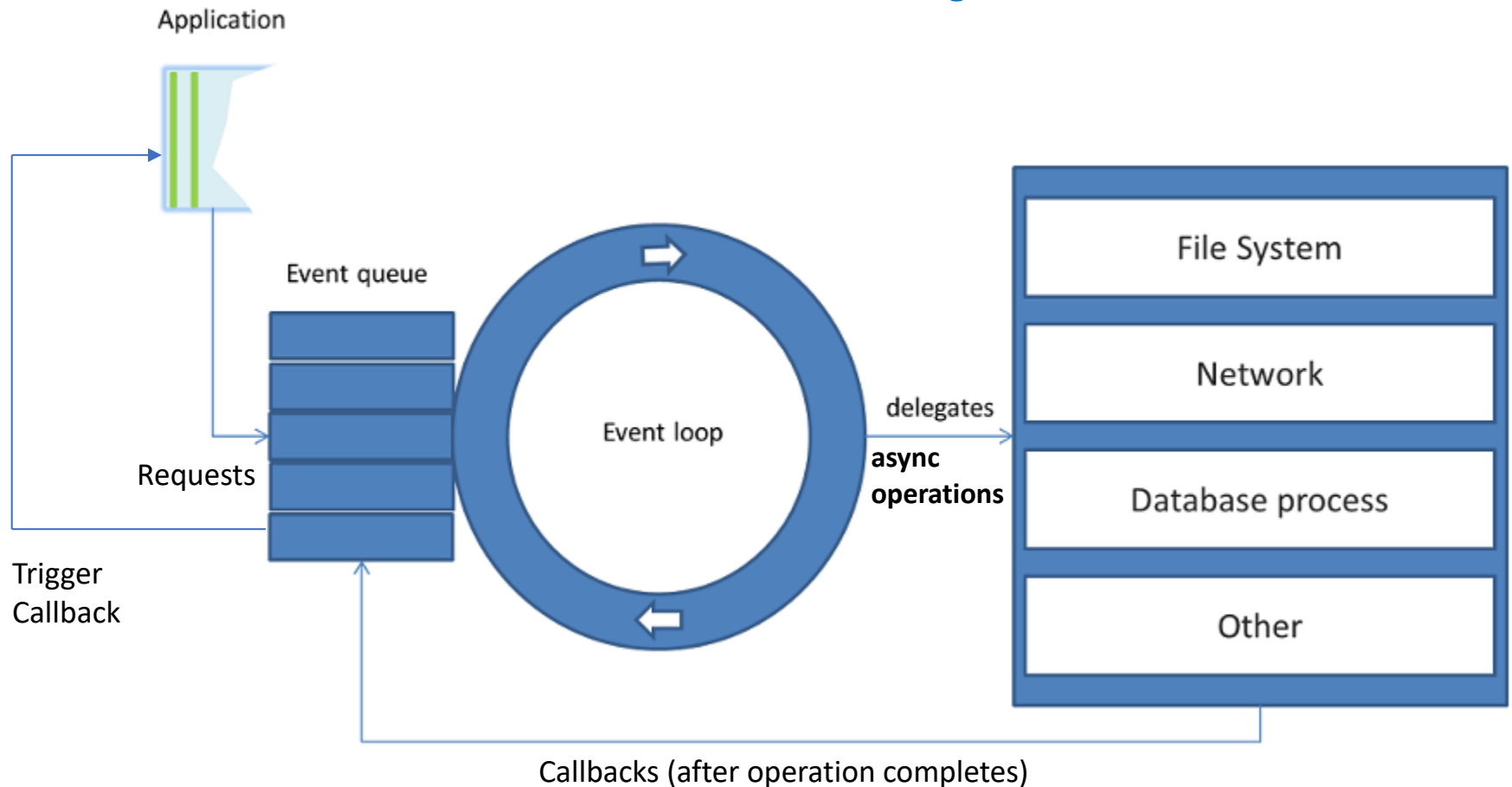
=> Async JavaScript programming using either:

- Callbacks
- Promises
- `async/await`

Key benefit of Async Programming = ***Responsiveness***

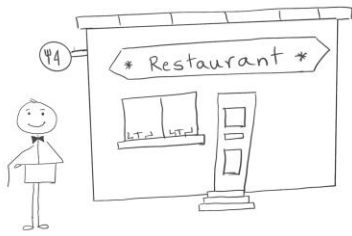
prevent blocking the main thread on long-running operations

Event Loop



Delegate async operations (e.g., I/O tasks) and manage callbacks to avoid blocking main thread

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



Analogy - Restaurant with a Single Waiter

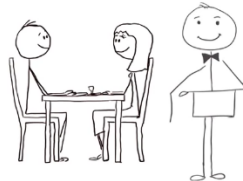


Table 1



Table 2



Table 3

Waiter
acts as
the **Event
Loop**



Event
Queue

When the dish is ready
then place a callback on
the events queue

```
let order1 = takeOrder(table1);  
chef.handOver(order, dish => serveTable(dish)); ✓
```

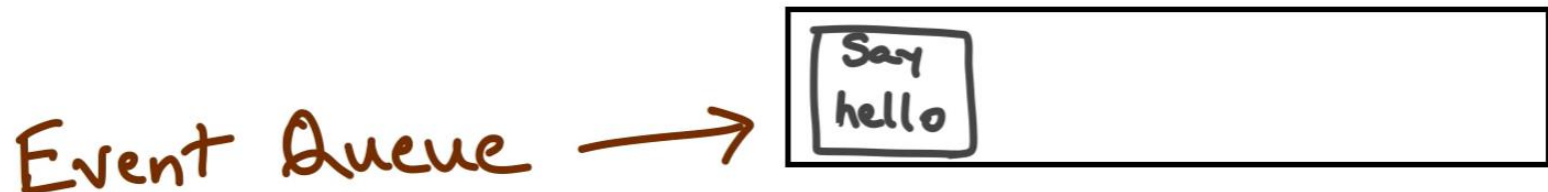
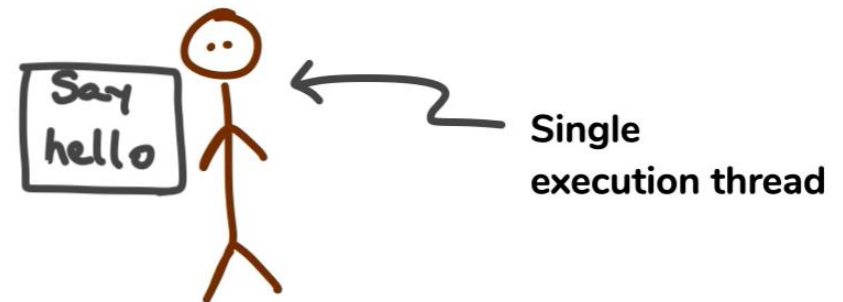
➡

```
let order2 = takeOrder(table2);
```

Event Queue & Event Loop

```
while isEmpty(eventQueue)  
  pull out first item from event queue  
  execute it
```

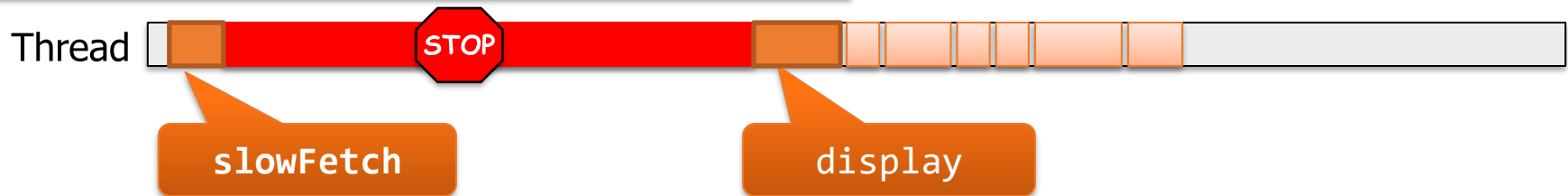
↑
Event
Loop



Synchronous vs. Asynchronous Functions

Synchronous → Wait for result before returning

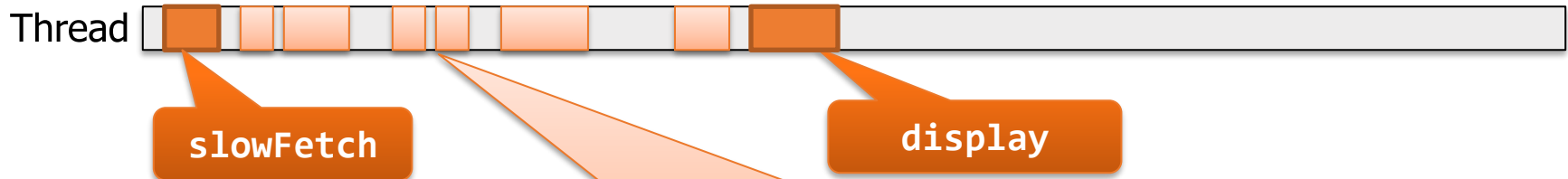
```
const result = slowFetch(...) // UI Thread  
display(result) // UI Thread
```



```
// Slow request with callbacks  
fun makeNetworkRequest(display) {  
  // The slow network request runs on another thread  
  slowFetch { result =>  
    // When the result is ready, this callback will get the result  
    display(result)  
  }  
}
```



Asynchronous → do an **asynchronous** call to slowFetch using background thread, then update UI with the result



UI not blocked

While waiting for a result, the main thread is **unblocked** so other functions can run



Callbacks

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

Callback Hell...

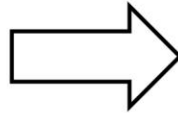
```
function getStockPrice(name, cb) {  
  getStockSymbol(name, (error, symbol) => {  
    if (error) {  
      cb(error);  
    }  
    else {  
      getStockPrice(symbol, (error, price) => {  
        if (error) {  
          cb(error);  
        }  
        else {  
          cb(price);  
      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



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
 - Is a way of promising that a work will be done (or might fail if the work could not be completed)
 - 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:

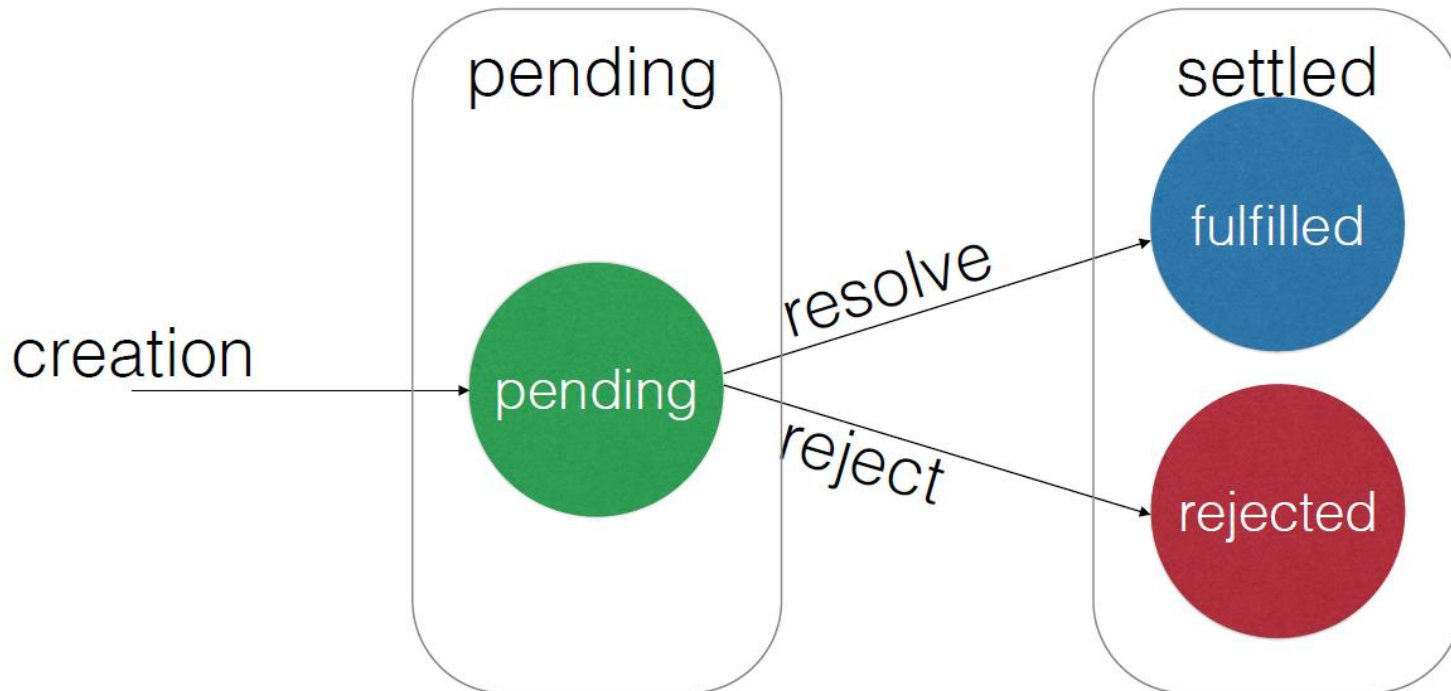
```
promise..then(onFulfilled)
```

```
.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



- **Pending** - Not settled yet
- **Fulfilled** - When a promise is resolved successfully.
- **Rejected** - When a promise failed.
- **Settled** - an *umbrella term* to describe that a promise is either fulfilled or rejected

How to create a Promise

```
const 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

```
const 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**

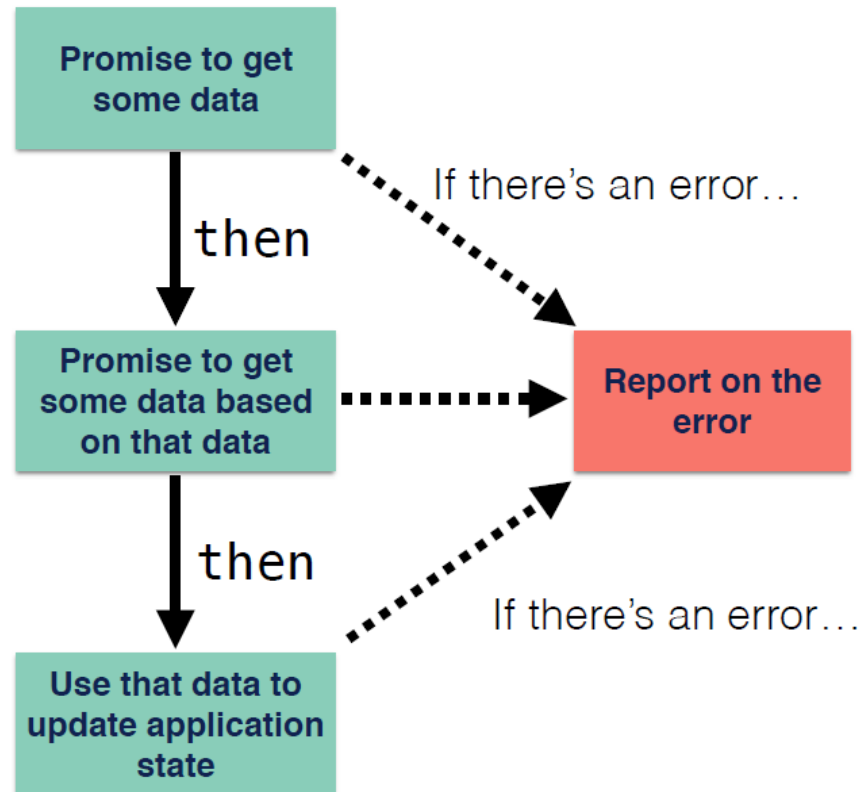
```
function getStockPrice(name) {  
    const symbol = getStockSymbol(name);  
    const price = getStockPrice(symbol);  
    return price;  
}
```

- **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))
```



What distinguishes promises?



1. Easier **exception handling**
2. Easier to **run promises in parallel** to improve the app performance
3. Easier **asynchronous** programming
 - Replace callback-based code with sequential **async** / **await** to handle asynchronous long-running tasks without blocking

Promise combinator methods

Promise combinator methods

- **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 (and ignores the remaining ones)

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

Differences

⚙️ Promise.all vs. Promise.allSettled

- `Promise.all` rejects as soon as a promise in the list is rejected.
- `Promise.allSettled` resolves regardless of rejected promise(s) within the list.

⚙️ Promise.race vs. Promise.any

- `Promise.race` short-circuits on the first settled (fulfilled or rejected) promise within the list.
- `Promise.any` short-circuits on the first fulfilled promise and continues to resolve regardless of rejected promises unless all within the list reject.

	Short-circuit?	Short-circuits on?	Fulfilled on?	Rejected on?
Promise.all	✓	First rejected promise	All promise fulfilled	First rejected promise
Promise.allSettled	✗	N/A	Always	N/A
Promise.race	✓	First settled	First promise fulfilled	First rejected promise
Promise.any	✓	First fulfilled	First promise fulfilled	All rejected promises



async / await

- Allows easier composition of promises compared to chaining using **.then**
 - Due to its sequential style, it's **easier** to understand

```
async function getStudent(studentId) {  
    const student = await getStudent(studentId);  
    student.courses = await getCourses(student.courseIds);  
    return student;  
}  
  
try {  
    const studentId = 2015002;  
    const student = getStudent(studentId);  
    console.log(JSON.stringify(student, null, 2));  
}  
catch(err) { console.log(err); }
```

How await works?

- When a function awaits a result of a call, it does NOT block instead the runtime:
 - **suspends** the function execution, removes it from the thread, and stores the state and the remaining function statements in memory until the result is ready then
 - **resumes** the function execution where it left off
- While it's suspended waiting for a result, **it unblocks the thread that it's running on**, so that the thread is free to be used for other tasks

Summary

- Async/await allows easier **asynchronous** programming
 - Replace callback-based code with sequential code to handle asynchronous long-running tasks without blocking
 - Structure of asynchronous code is the same as synchronous code