Asynchronous Patterns in JavaScript

Callbacks

Promises

Async/Await

Synchronous vs. Asynchronous

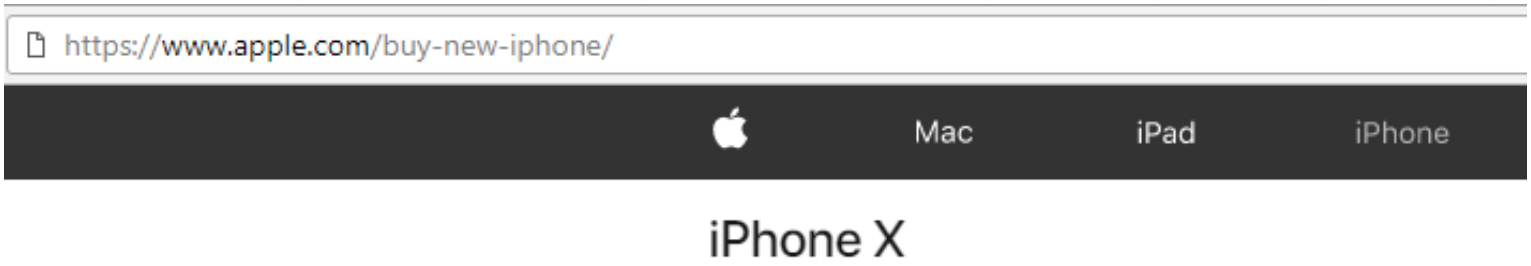
Buying newly released iPhone

- **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, and 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

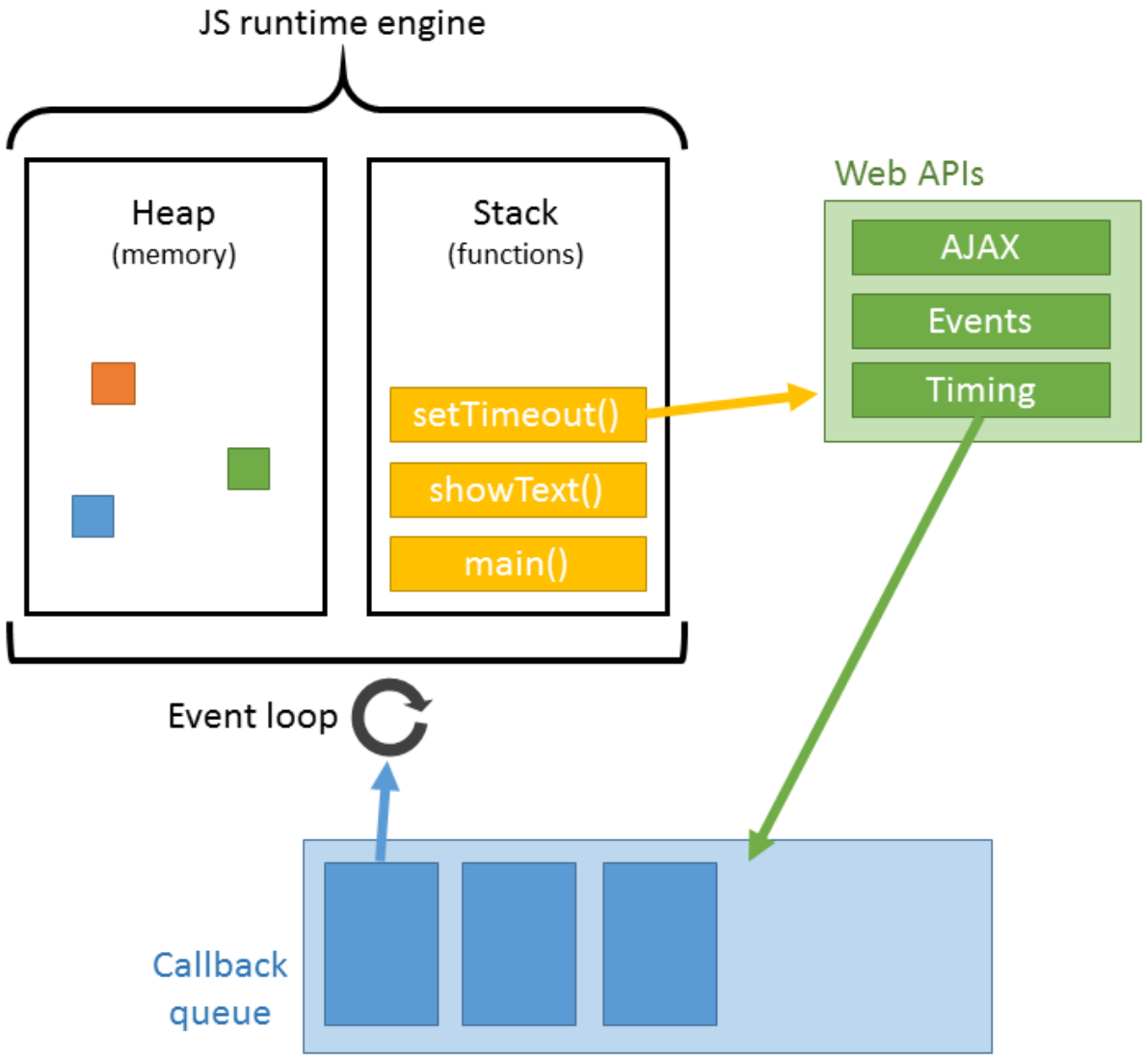
- CPU demanding tasks delay execution of all other tasks => **UI may become unresponsive**
- Accessing resources such as files blocks the entire program
  - Especially problematic with web resources
    - Resource may be large
    - Server may hang
    - Slow connection means slow loading causing UI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

JavaScript Event Loop



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 is a function that is passed to another function as a parameter:
  - The other function can call the passed one
  - The other function can give arguments
- Examples of callbacks:
  - Event handlers are sort-of callbacks
  - setTimeout and setInterval take 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.getElementById("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);
                }
            })
        }
    })
}
```

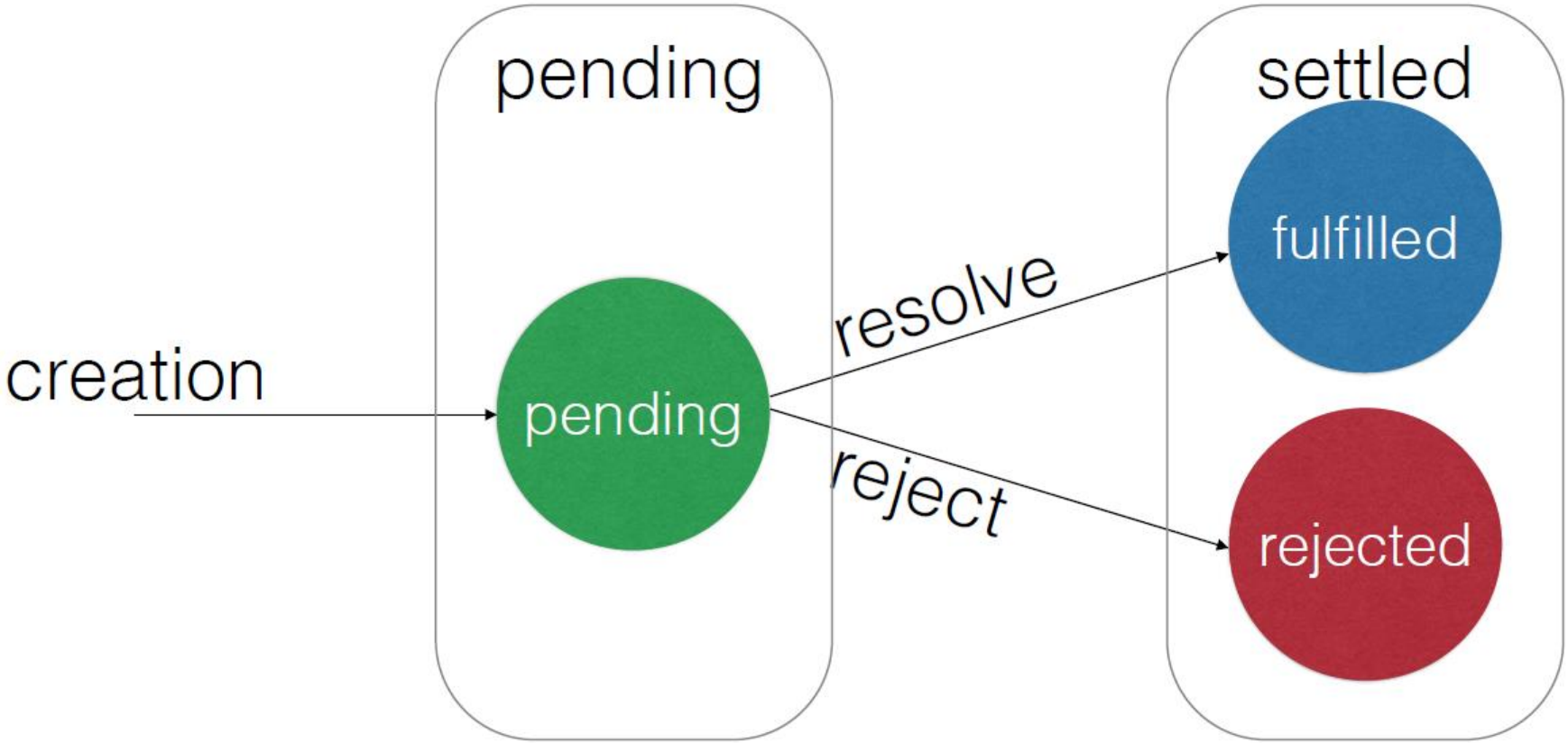
Promises

- 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);
```

  - 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

- loadImage returns a promise to load a given image

```
function loadImage(url){
    return new Promise(function(resolve, reject) {
        var img  = new Image();
        img.src=url;
        img.onload = function(){
            resolve(img);
        }
        img.onerror = function(e){
            reject(e);
        }
    });
}
```

Once the image is loaded, we'll resolve the promise

If the image has an error, the promise is rejected

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**

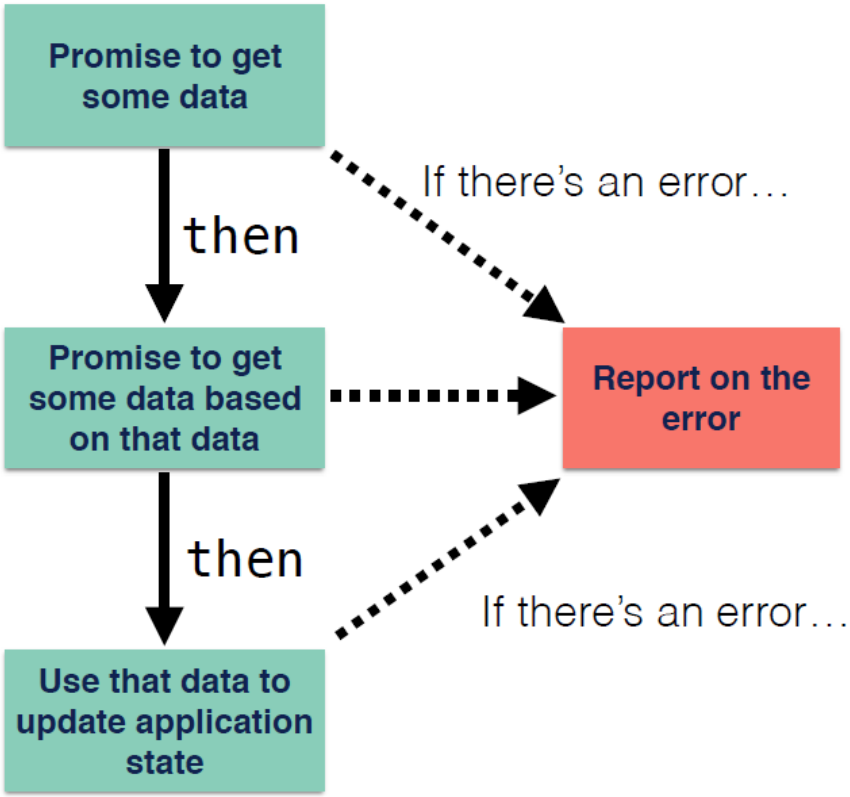
```
function getStockPrice(name) {
    let symbol = getStockSymbol(name);
    let price = getStockPrice(symbol);
    return price;
}
```

- **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
=> { ... });
```

ES7 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);
    let courses = await getCourses(student.courseIds);
    student.courses = courses;
    return student;
}

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