ES2015

THE SHAPE of JAVASCRIPT to COME

Modules – Part I

Level 5 – Section 2

Polluting the Global Namespace

The common solution for modularizing code relies on using **global variables**. This increases the chances of **unexpected side effects** and potential **naming conflicts**.

index.html

```
<!DOCTYPE html>
<body>
  <script src="./jquery.js"></script>
  <script src="./underscore.js"></script>
  <script src="./flash-message.js"></script>
</body>
```

Libraries add to the global namespace

```
let element = $("...").find(...);
let filtered = _.each(...);
flashMessage("Hello");
```

Global variables can cause naming conflicts

Creating Modules

ES2015

Let's create a new JavaScript module for displaying flash messages.

flash-message.js

flash-message.js

```
export default function(message){
  alert(message);
}
```

The export keyword exposes this function to the module system

The default type export is the simplest way to export a function

Importing Modules With Default Export

ES2015

To import modules we use the *import* keyword, specify a new local variable to hold its content, and use the *from* keyword to tell the JavaScript engine where the module can be found.

flash-message.js

app.js

app.js

```
import flashMessage from './flash-message';

flashMessage("Hello");
```

Can be named anything because it's default export

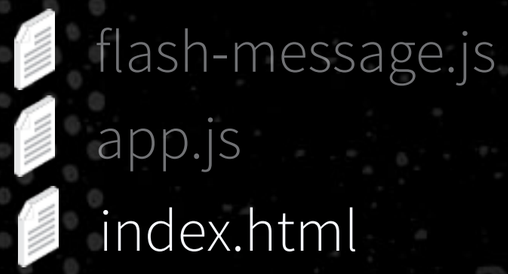
File name with a .js extension must be on the same folder

flash-message.js

```
export default function(message){
  alert(message);
}
```

Running Code From Modules

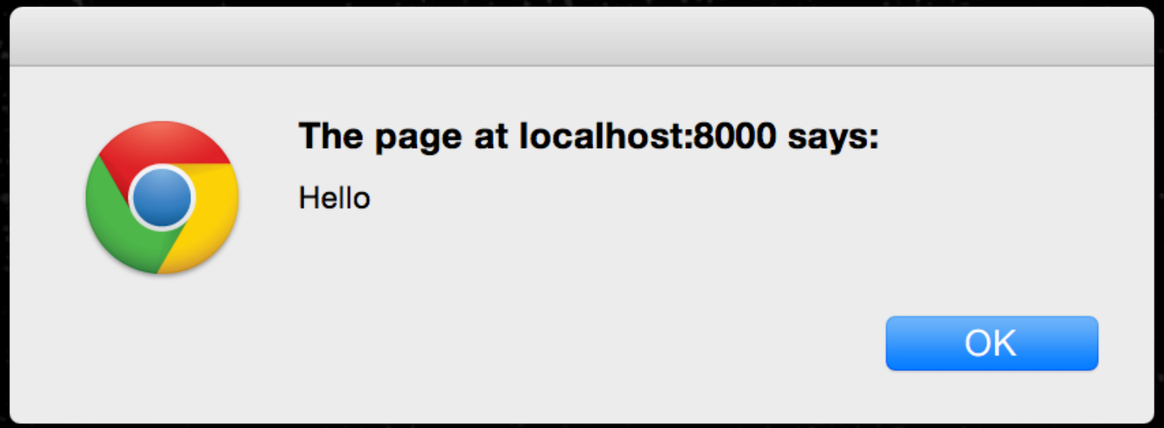
Modules still need to be imported via <*script*>, but **no longer pollute the global namespace.**



index.html

```
<!DOCTYPE html>
<body>
  <script src="./flash-message.js"></script>
  <script src="./app.js"></script>
</body>
```

Not adding to the global namespace



Can’t Default Export Multiple Functions

The *default* type export **limits** the number of functions we can export from a module.

flash-message.js
app.js
index.html

flash-message.js

```
export default function(message){
  alert(message);
}

function logMessage(message){
  console.log(message);
}
```

Not available outside this module

Using Named Exports

In order to **export multiple functions** from a single module, we can use the **named** export.

flash-message.js

app.js

index.html

No longer using default type export

flash-message.js

```
export function alertMessage(message){
  alert(message);
}

export function logMessage(message){
  console.log(message);
}
```

Importing Named Exports

Functions from **named** exports must be assigned to variables with **the same name** enclosed in curly braces.

- flash-message.js
- app.js
- index.html

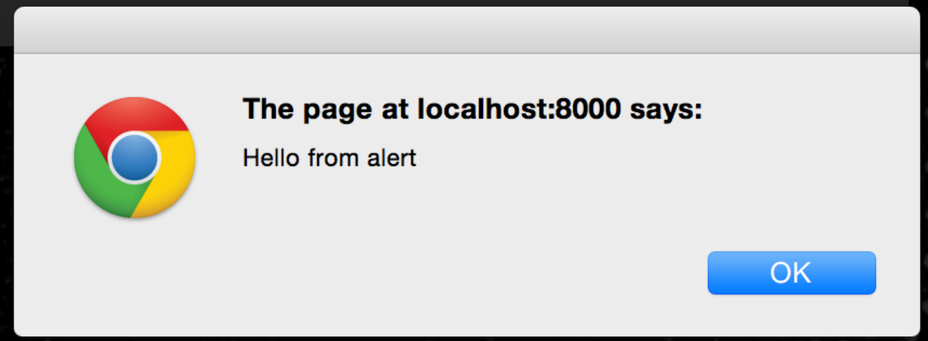
app.js

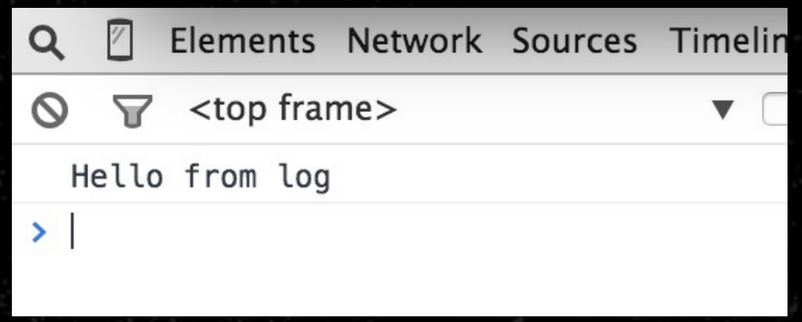
```
import { alertMessage, logMessage } from './flash-message';

alertMessage('Hello from alert');
logMessage('Hello from log');
```

Names must match

flash-message.js

```
export function alertMessage(message){
  alert(message);
}

export function logMessage(message){
  console.log(message);
}
```





Importing a Module as an Object

We can also **import the entire module** as an object and call each function as a **property** from this object.

- flash-message.js
- app.js
- index.html

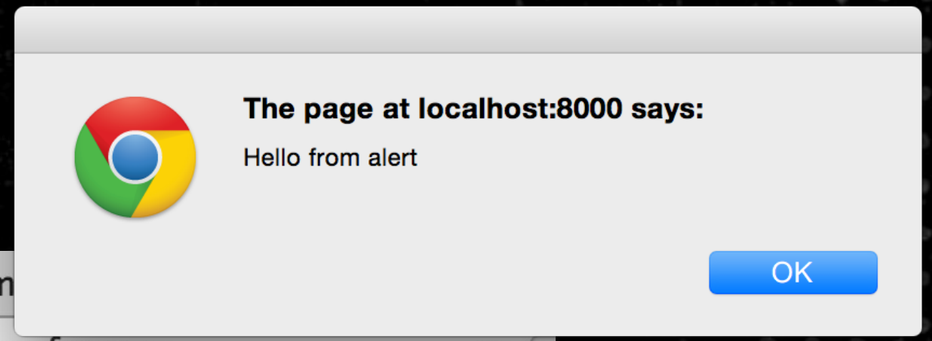
app.js

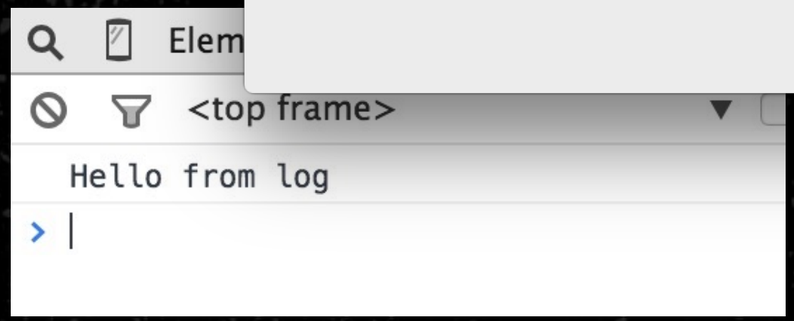
```
import * as flash from './flash-message';

flash.alertMessage('Hello from alert');
flash.logMessage('Hello from log');
```

functions become object properties

flash-message.js

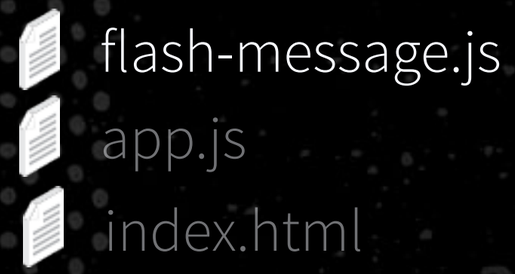
```
export function alertMessage(message){
  alert(message);
}

export function logMessage(message){
  console.log(message);
}
```





Removing Repeated Export Statements

We are currently calling export statements every time we want to export a function.



flash-message.js

```
export function alertMessage(message){
  alert(message);
}

export function logMessage(message){
  console.log(message);
}
```

One export call for each function that we want to expose from our module

Exporting Multiple Functions at Once

We can export multiple functions at once by passing them to *export* inside curly braces.

flash-message.js

app.js

export can take multiple function names between curly braces

Imported just like before

flash-message.js

```
function alertMessage(message){
  alert(message);
}

function logMessage(message){
  console.log(message);
}

export { alertMessage, logMessage }
```

app.js

```
import { alertMessage, logMessage } from './flash-message';

alertMessage('Hello from alert');
logMessage('Hello from log');
```

Modules – Part II

Level 5 – Section 3

Extracting Hardcoded Constants

Redefining constants across our application is **unnecessary repetition** and can lead to **bugs**.

We define our constants here...

load-profiles.js

```
function loadProfiles(userNames){
  const MAX_USERS = 3;
  if(userNames.length > MAX_USERS){
    //...
  }

  const MAX_REPLIES = 3;
  if(someElement > MAX_REPLIES){
    //...
  }
}

export { loadProfiles }
```

...and then we define them again on different places

list-replies.js

```
function listReplies(replies=[]){
  const MAX_REPLIES = 3;
  if(replies.length > MAX_REPLIES){
    //...
  }
}
export { listReplies }
```

display-watchers.js

```
function displayWatchers(watchers=[]){
  const MAX_USERS = 3;
  if(watchers.length > MAX_USERS){
    //...
  }
}
export { displayWatchers }
```

Exporting Constants

Placing constants on their own module allows them to be reused across other modules and **hides implementation details** (a.k.a., **encapsulation**).

constants.js

constants.js

```
export const MAX_USERS = 3;
export const MAX_REPLIES = 3;
```

Either syntax works

constants.js

```
const MAX_USERS = 3;
const MAX_REPLIES = 3;

export { MAX_USERS, MAX_REPLIES };
```

How to Import Constants

To *import* constants, we can use the exact same syntax for importing functions.


constants.js


load-profiles.js

Details are encapsulated inside of the constants module

load-profiles.js

```
import { MAX_REPLIES, MAX_USERS } from './constants';

function loadProfiles(userNames){

  if(userNames.length > MAX_USERS){
    //...
  }

  if(someElement > MAX_REPLIES){
    //...
  }
}
```

Importing Constants

We can now import and use our constants from other places in our application.


constants.js

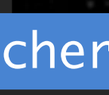

load-profiles.js


list-replies.js


display-watchers.js

list-replies.js 

```
import { MAX_REPLIES } from './constants';

function listReplies(replies = []){
  if(replies.length > MAX_REPLIES){
    //...
  }
}
```

display-watchers.js 

```
import { MAX_USERS } from './constants';

function displayWatchers(watchers = []){
  if(watchers.length > MAX_USERS){
    //...
  }
}
```

Exporting Class Modules With Default Export

Classes can also be exported from modules using the same syntax as functions. Instead of 2 individual functions, we now have **2 instance methods** that are part of a class.

flash-message.js

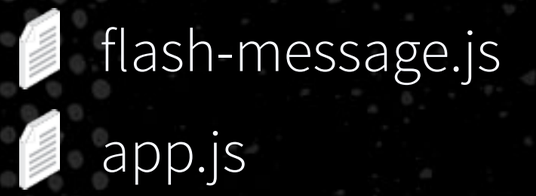
default allows this class to be set to any variable name once it's imported

flash-message.js

```
export default class FlashMessage {

  constructor(message){
    this.message = message;
  }

  renderAlert(){
    alert(`${this.message} from alert`);
  }

  renderLog(){
    console.log(`${this.message} from log`);
  }
}
```

Using Class Modules With Default Export

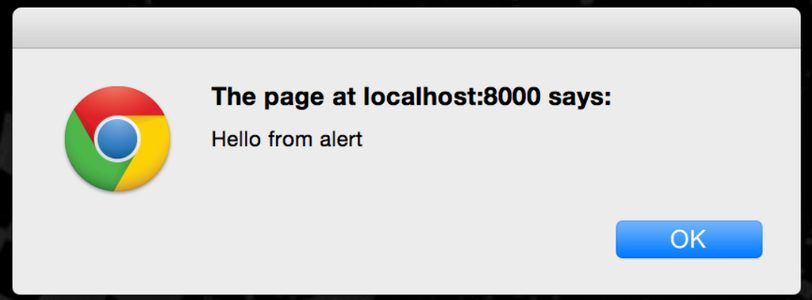
Imported classes are assigned to a variable using *import* and can then be used to create **new instances**.



flash-message.js

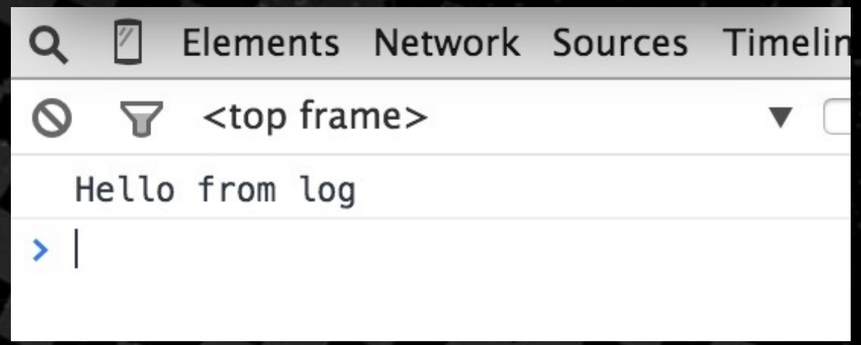
```
export default class FlashMessage {
  // ...
}
```

Exporting a class, so F is capitalized



app.js

```
import FlashMessage from './flash-message';

let flash = new FlashMessage("Hello");
flash.renderAlert();
flash.renderLog();
```



Creates instance and calls instance methods

Using Class Modules With Named Export

Another way to export classes is to first define them, and then use the *export* statement with the class name inside curly braces.

flash-message.js

Plain old JavaScript class declaration

```
class FlashMessage {

  constructor(message){
    this.message = message;
  }

  renderAlert(){
    alert(`${this.message} from alert`);
  }
  renderLog(){
    console.log(`${this.message} from log`);
  }
}

export { FlashMessage }
```

Exports class to the outside world

Using Class Modules With Named Export

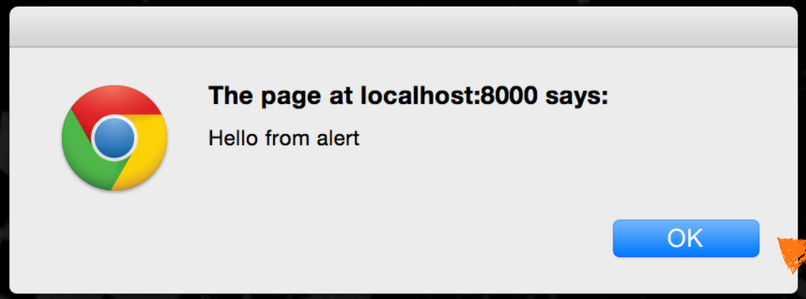
When using **named export**, the script that loads the module needs to assign it to a variable with **the same name as the class.**

flash-message.js

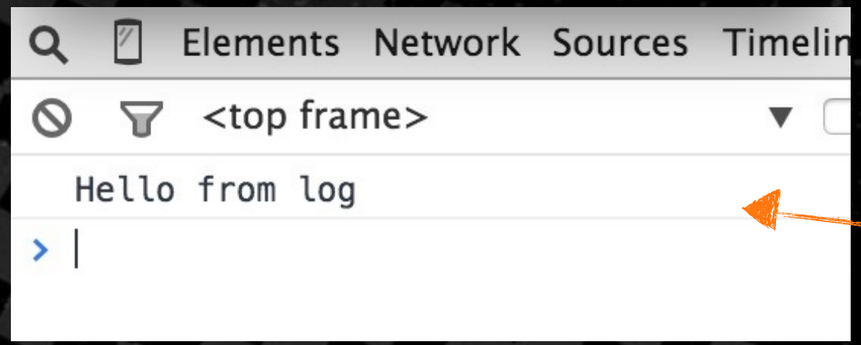
app.js

```
class FlashMessage {
  //...
}

export { FlashMessage }
```

Names must match




```
import { FlashMessage } from './flash-message';

let flash = new FlashMessage("Hello");
flash.renderAlert();
flash.renderLog();
```



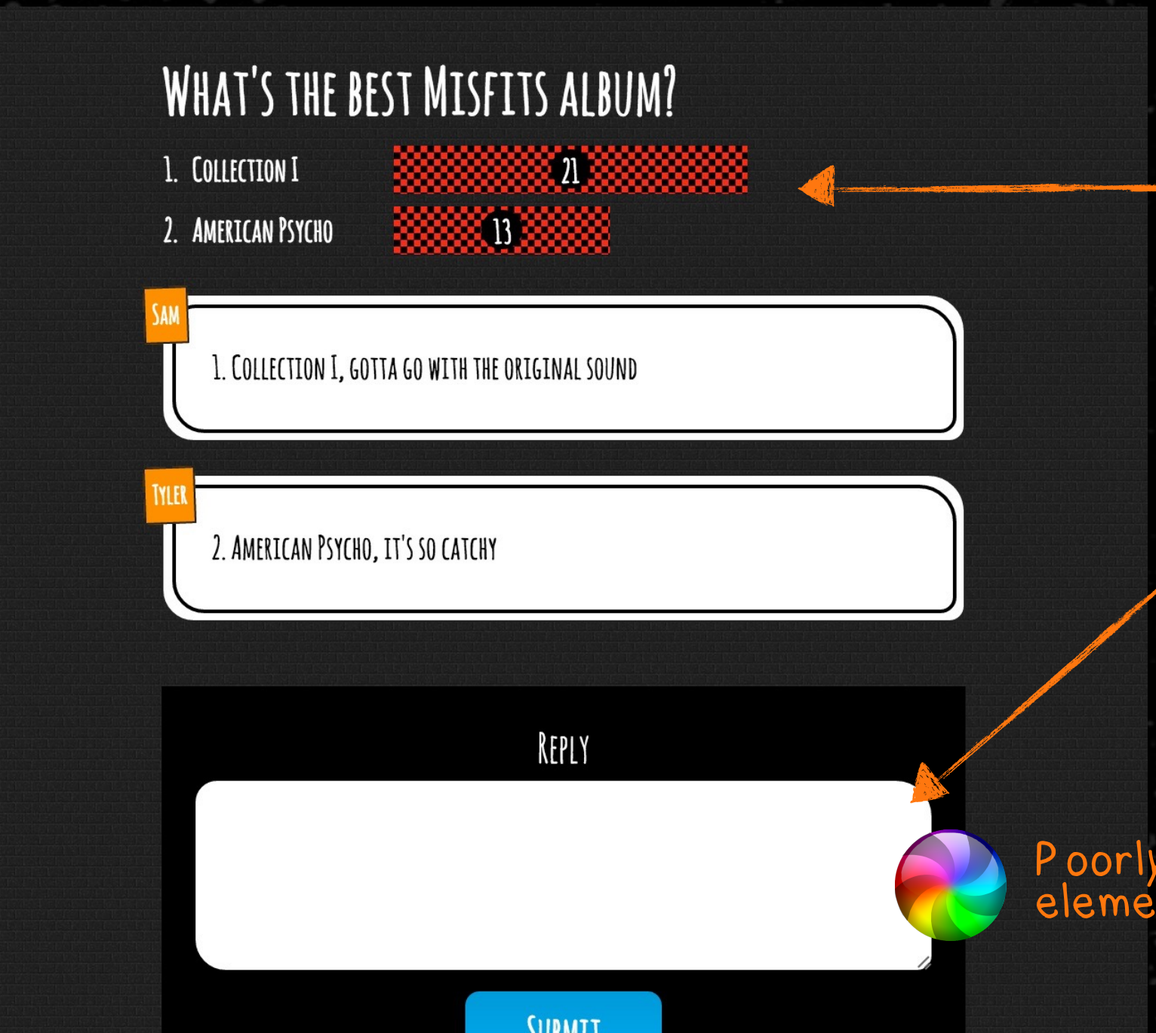
Same result as before

Promises

Level 6 – Section 1

Fetching Poll Results From the Server

It's very important to understand how to work with JavaScript's **single-thread model**. Otherwise, we might accidentally **freeze** the entire app, to the detriment of user experience.



A script requests poll results from the server...

...and while the script waits for the response, users must be able to interact with the page.

Poorly written code can make other elements on the page unresponsive

Avoiding Code That Blocks

Once the browser **blocks** executing a script, it stops running other scripts, rendering elements, and responding to user events like keyboard and mouse interactions.

Synchronous style functions wait for return values

```
let results = getPollResultsFromServer("Sass vs. LESS");
ui.renderSidebar(results);
```

Page freezes until a value is returned from this function

In order to **avoid blocking** the main thread of execution, we write non-blocking code like this:

Asynchronous style functions pass callbacks

```
getPollResultsFromServer("Sass vs. Less", function(results){
  ui.renderSidebar(results);
});
```

Passing Callbacks to Continue Execution

In **continuation-passing style** (CPS) async programming, we tell a function how to continue execution by passing callbacks. It can grow to **complicated nested code**.

```
getPollResultsFromServer(pollName, function(error, results){
  if(error){ //.. handle error }
  //...
  ui.renderSidebar(results, function(error){
    if(error){ //.. handle error }
    //...
    sendNotificationToServer(pollName, results, function(error, response){
      if(error){ //.. handle error }
      //...
      doSomethingElseNonBlocking(response, function(error){
        if(error){ //.. handle error }
        //...
      });
  });
});
```

When nested callbacks start to grow, our code becomes harder to understand

The Best of Both Worlds With Promises

ES2015

A Promise is a new abstraction that allows us to write async code in an easier way.

```
getPollResultsFromServer("Sass vs. LESS")
  .then(ui.renderSidebar)
  .then(sendNotificationToServer)
  .then(doSomethingElseNonBlocking)
  .catch(function(error){
      console.log("Error: ", error);
  });
```

Still non-blocking, but not using nested callbacks anymore

*Let's learn how to create **Promises!***

Creating a New Promise Object

ES2015

The Promise constructor function takes an anonymous function with 2 callback arguments known as **handlers**.

```
function getPollResultsFromServer(pollName){

  return new Promise(function(resolve, reject){
    //...
    resolve(someValue);

    //...
    reject(someValue);
  });
};
```

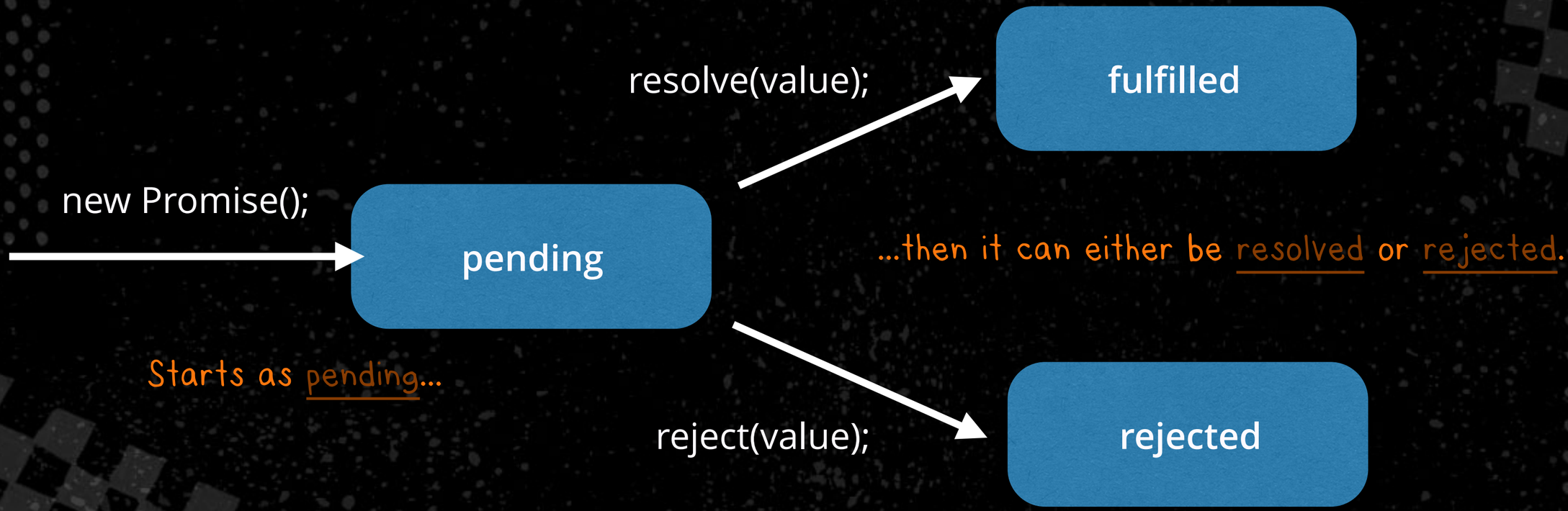
Handlers are responsible for either resolving or rejecting the Promise

Called when the non-blocking code is done executing

Called when an error occurs

The Lifecycle of a Promise Object

Creating a new Promise automatically sets it to the **pending** state. Then, it can do 1 of 2 things: become **fulfilled** or **rejected**.



Returning a New Promise Object

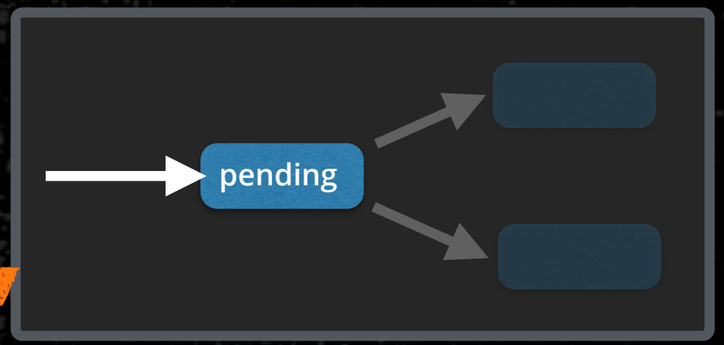
A Promise represents a **future value**, such as the eventual result of an **asynchronous** operation.

```
let fetchingResults = getPollResultsFromServer("Sass vs. Less");
```

Not the actual result, but a Promise object

No longer need to pass a callback function as argument

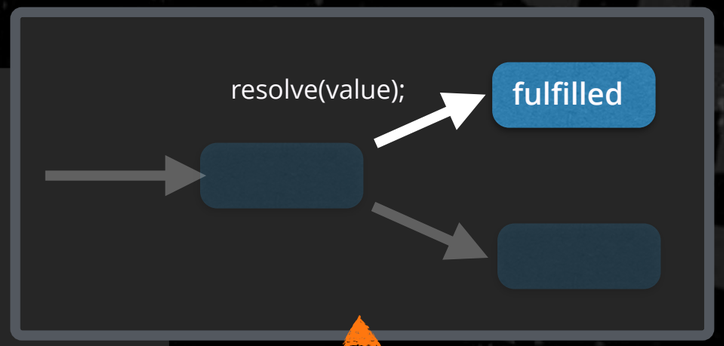
Starts on pending state



Resolving a Promise

Let's wrap the *XMLHttpRequest* object API within a Promise. Calling the *resolve()* handler moves
the Promise to a **fulfilled** state.

```
function getPollResultsFromServer(pollName){

  return new Promise(function(resolve, reject){
    let url = `/results/${pollName}`;
    let request = new XMLHttpRequest();
    request.open('GET', url, true);
    request.onload = function() {
      if (request.status >= 200 && request.status < 400) {
        resolve(JSON.parse(request.response));
      }
    };
    //...
    request.send();
  });
};
```



Resolving a Promise moves it to a fulfilled state

We call the resolve() handler upon a successful response

Reading Results From a Promise

We can use the *then()* method to read results from the Promise once it's resolved. This method takes a function that will only be invoked once the Promise is **resolved**.

```
let fetchingResults = getPollResultsFromServer("Sass vs. Less");
fetchingResults.then(function(results){
  ui.renderSidebar(results);
});
```

This function renders HTML to the page

This is the argument previously passed to resolve()

```
function getPollResultsFromServer(pollName){
    //...
        resolve(JSON.parse(request.response));
    //...
};
```

Removing Temporary Variables

We are currently using a **temporary variable** to store our Promise object, but it's not really necessary. Let's replace it with **chaining function calls**.

```
let fetchingResults = getPollResultsFromServer("Sass vs. Less");
fetchingResults.then(function(results){
  ui.renderSidebar(results);
});
```

Temporary variable is unnecessary

Same as this

```
getPollResultsFromServer("Sass vs. Less")
  .then(function(results){
    ui.renderSidebar(results);
  });
```

Chaining Multiple Thens

We can also chain multiple calls to *then()* — the **return value** from 1 call is passed as argument to the next.

```
getPollResultsFromServer("Sass vs. Less")
  .then(function(results){
    return results.filter((result) => result.city === "Orlando");
  })
  .then(function(resultsFromOrlando){
    ui.renderSidebar(resultsFromOrlando);
  });
```

Only returns poll results from Orlando

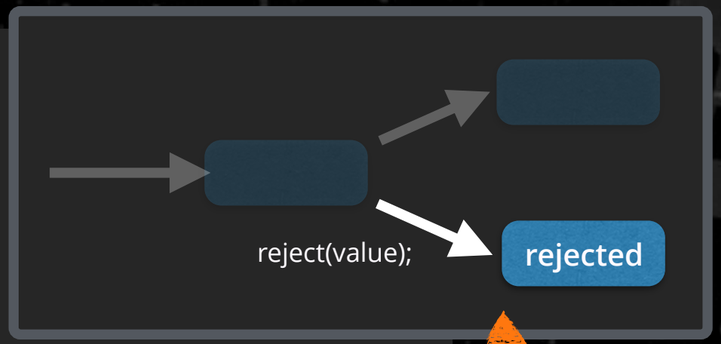
The return value from one call to then...

...becomes the argument to the following call to then.

Rejecting a Promise

We'll call the *reject()* handler for **unsuccessful status codes** and also when the *onerror* event is triggered on our request object. Both move the Promise to **a rejected state**.

```
function getPollResultsFromServer(pollName){

  return new Promise(function(resolve, reject){
   //...
   request.onload = function() {
      if (request.status >= 200 && request.status < 400) {
        resolve(request.response);
      } else {
        reject(new Error(request.status));
      }
    };
    request.onerror = function() {
      reject(new Error("Error Fetching Results"));
    };
    //...
```



Rejecting a Promise moves it to a rejected state

We call the reject() handler, passing it a new Error object

Catching Rejected Promises

Once an error occurs, execution moves immediately to the *catch()* function. None of the remaining *then()* functions are invoked.

```
getPollResultsFromServer("Sass vs. Less")
  .then(function(results){
    return results.filter((result) => result.city === "Orlando");
  })
  .then(function(resultsFromOrlando){
    ui.renderSidebar(resultsFromOrlando);
  })
  .catch(function(error){
      console.log("Error: ", error);
  });
```

When an error occurs here...

...then none of these run...

...and execution moves straight here.

Passing Functions as Arguments

We can make our code more succinct by passing function arguments to *then*, instead of using anonymous functions.

```
function filterResults(results){ //... }

let ui = {
  renderSidebar(filteredResults){ //... }
};

getPollResultsFromServer("Sass vs. Less")
  .then(filterResults)
  .then(ui.renderSidebar)
  .catch(function(error){
      console.log("Error: ", error);
  });
```

Remember the new method initializer shorthand syntax?

Passing function arguments make this code easier to read

Still catches all errors from previous calls

Iterators

Level 6 – Section 2

What We Know About Iterables So Far

Arrays are **iterable** objects, which means we can use them with *for...of*.

```
let names = ["Sam", "Tyler", "Brook"];

for(let name of names){
  console.log( name );
}
```

```
> Sam
> Tyler
> Brook
```

Plain JavaScript objects are **not iterable**, so they **do not work** with *for...of* out-of-the-box.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

for(let p of post){
  console.log(p);
}
```

```
> TypeError: post[Symbol.iterator] is not a function
```

Iterables Return Iterators

Iterables return an **iterator** object. This object knows how to **access items from a collection** 1 at a time, while **keeping track of its current position** within the sequence.

```
let names = ["Sam", "Tyler", "Brook"];

for(let name of names){
  console.log( name );
}
```

What's really happening behind the scenes

```
let iterator = names[Symbol.iterator]();

let firstRun = iterator.next();
let name = firstRun.value;

let secondRun = iterator.next();
let name = secondRun.value;

let thirdRun  = iterator.next();
let name = thirdRun.value;

let fourthRun = iterator.next();
```

```
{ done: false, value: "Sam" }

{ done: false, value: "Tyler" }

{ done: false, value: "Brook" }
```

Breaks out of the loop when done is true

```
{ done: true, value: undefined }
```

Understanding the next Method

Each time *next()* is called, it returns an object with **2** specific properties: *done* and *value.*

```
let names = ["Sam", "Tyler", "Brook"];

for(let name of names){
  console.log( name );
}
```

Here's how values from these 2 properties work:

done (boolean)

- Will be *false* if the iterator is able to return a value from the collection
- Will be *true* if the iterator is past the end of the collection

value (any)

- Any value returned by the iterator. When *done* is *true*, this returns *undefined*.

The First Step Toward an Iterator Object

An iterator is an object with a *next* property, returned by the result of calling the *Symbol.iterator* method.

```
let post = {
  title: "New Features in JS",
  replies: 19
};
```

```
post[Symbol.iterator] = function(){

  let next = () => {

  }

  return { next };
};
```

Iterator object

```
for(let p of post){
  console.log(p);
}
```

> Cannot read property 'done' of undefined

Different error message... We are on the right track!

Navigating the Sequence

We can use *Object.keys* to build an array with property names for our object. We'll also use a counter (*count*) and a boolean flag (*isDone*) to help us navigate our collection.

```
let post = { //... };

post[Symbol.iterator] = function(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  let next = () => {

  }

  return { next };
};
```

Returns an array with property names

Allows us to access the properties array by index

Will be set to true when we are done with the loop

Returning done and value

We use *count* to keep track of the sequence and also to fetch values from the *properties* array.

```
let post = { //... };

post[Symbol.iterator] = function(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  let next = () => {
    if(count >= properties.length){
      isDone = true;
    }
    return { done: isDone, value: this[properties[count++]] };
  }
  return { next };
};
```

Ends the loop after reaching the last property

Fetches the value for the next property

++ only increments count after it's read

this refers to the post object

Running Our Custom Iterator

We've successfully made our plain JavaScript object **iterable**, and it can now be used with *for...of*.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){
  //...
  return { next };
};

for(let p of post){
  console.log(p);
}
```

```
> New Features in JS
> 19
```

Iterables With the Spread Operator

Objects that comply with the iterable protocol can also be used with the **spread operator**.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){
  //...
  return { next };
};

let values = [...post];
console.log( values );
```

```
> ['New Features in JS', 19]
```

Groups property values and returns an array

Iterables With Destructuring

Lastly, **destructuring** assignments will also work with iterables.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){
  //...
  return { next };
};
```

```
let [title, replies] = post;
console.log( title );
console.log( replies );
```

```
> New Features in JS
> 19
```

Generators

Level 6 – Section 3

Generator Functions

ES2015

The *function* * declaration defines **generator functions**. These are special functions from which we can use the *yield* keyword to return **iterator** objects.

It's a star character!

```
function *nameList(){
  yield "Sam";
  yield "Tyler";
}
```

```
{ done: false, value: "Sam" }
{ done: false, value: "Tyler" }
```

```
function *nameList(){
function* nameList(){
function * nameList(){
```

Doesn't matter where we place the star, as long as it's the first thing after the function keyword

Generator Objects and *for...of*

Generator functions return objects that provide the same *next* method expected by *for...of*, the *spread operator*, and the *destructuring assignment*.

```
function *nameList(){
  yield "Sam";
  yield "Tyler";
}
```

```
{ done: false, value: "Sam" }
{ done: false, value: "Tyler" }
```

Calling the function returns a generator object

```
for(let name of nameList()){
  console.log( name );
}
```

```
> Sam
> Tyler
```

```
let names = [...nameList()];
console.log( names );
```

```
> ["Sam", "Tyler"]
```

```
let [first, second] = nameList();
console.log( first, second );
```

```
> Sam Tyler
```

Replacing Manual Iterator Objects

Knowing how to **manually** craft an iterator object is important, but there's a **shorter** syntax.

```
let post = { title: "New Features in JS", replies: 19 };

post[Symbol.iterator] = function(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  let next = () => {
    if(count >= properties.length){
      isDone = true;
    }
    return { done: isDone, value: this[properties[count++]] };
  }
  return { next };
}
```

We can make this shorter

Refactoring to Generator Functions

ES2015

Each time *yield* is called, our function returns a **new iterator** object and then **pauses** until it's called again.

```
let post = { title: "New Features in JS", replies: 19 };

post[Symbol.iterator] = function *(){

  let properties = Object.keys(this);
  for(let p of properties){
    yield this[p];
  }
}
```

Generator function signature

first 'title'
then 'replies'

Same as manually returning each property value

```
post[Symbol.iterator] = function *(){
  yield this.title;
  yield this.replies;
}
```

```
for(let p of post){
  console.log( p );
}
```

Successfully returns correct values!

```
> New Features in JS
> 19
```