

ES6

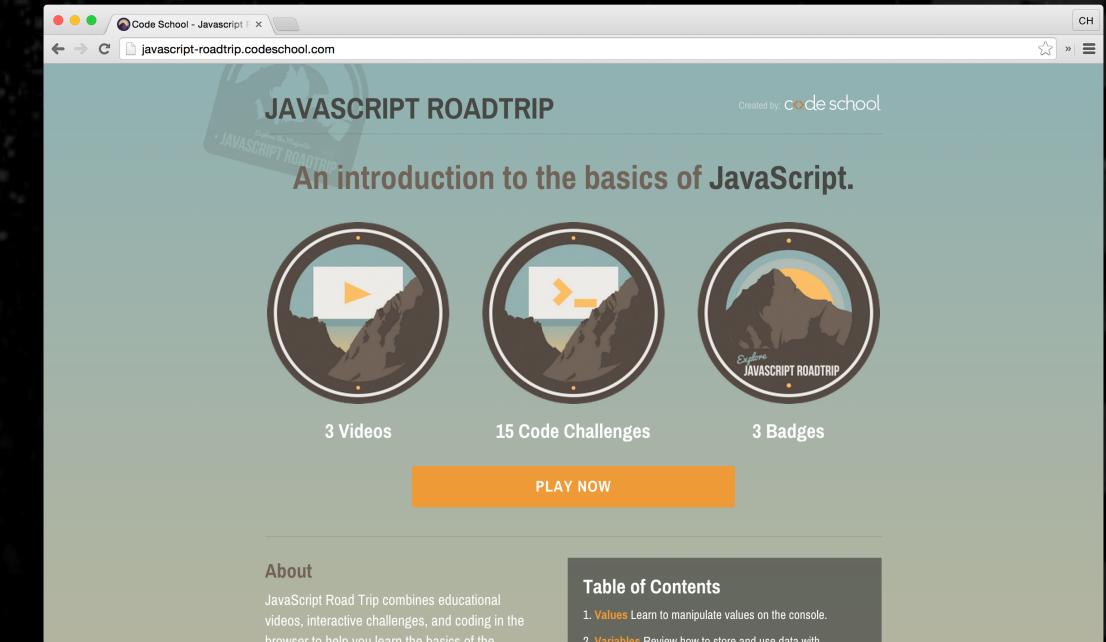
THE **SHAPE** of **JAVASCRIPT** to **COME**



New to JavaScript? Go here first!
<http://javascript-roadtrip.codeschool.com>

Prerequisites

Previous experience building JavaScript applications is necessary. For those brand new to the language, you can check out our intro course, *JavaScript Road Trip*.



What Is ES2015 (formerly known as ES6) ?

Previously called *ES6*, *ES2015* is the most extensive update to the JavaScript language since the publication of its first edition in 1997.

Kind of a big deal.

This Course

We'll cover some of the new features in JavaScript, including:

- *Variable* and *function* declarations
- More efficient *data structures*
- *Promises* and *Generators*
- ...and more

Declarations with *let*

Level 1 – Section 1

Forum Web App

The screenshot shows a forum application window titled "The Forum" with the URL "localhost:8000". At the top, there's a logo with the word "FORUM" in a stylized font. Below it, three threads are listed in a vertical stack:

- IN HYBRID MOMENTS, GIVE ME A MOMENT**
LAST REPLY ON JULY 15, 1997
- SOUND SYSTEM GONNA BRING ME BACK UP**
LAST REPLY ON JULY 1, 1991
- WHEN I'M OUT WALKIN', I STRUT MY STUFF**
LAST REPLY ON APRIL 19, 1983

The screenshot shows a forum application window titled "The Forum" with the URL "localhost:8000". At the top, there's a logo with the word "FORUM" in a stylized font. Below it, a single thread is displayed:

WHAT'S THE BETTER MISFITS ALBUM?

1. STATIC AGE
2. COLLECTION I

Below the list, a user named CARLOS S. has posted a reply:

CARLOS S.
1. COLLECTION I, GOTTA GO WITH THE ORIGINAL SOUND

At the bottom, another user named SAMANTHA A. has a partially visible reply:

SAMANTHA A.
2. AMERICAN PSYCHO. IT'S SO CATCHY

The screenshot shows a detailed view of a forum thread titled "IN HYBRID MOMENTS, GIVE ME A MOMENT". The thread has three replies:

- CARLOS S.**
AM I FOLLOWING ALL THE RIGHT LEADS? OR AM I ABOUT TO GET LOST IN SPACE? WHEN MY TIME COMES THEY'LL WRITE MY DESTINY. WILL YOU TAKE THIS RIDE?
- SAMANTHA A.**
GOOD FEELING, WON'T YOU STAY WITH ME JUST A LITTLE LONGER? IT ALWAYS SEEMS LIKE YOU'RE LEAVING WHEN I NEED YOU HERE JUST A LITTLE LONGER
- JON F.**
STATIC PULSE INSIDE OF MUSIC BRINGING US ESCAPE. IT'S ALWAYS TEMPORARY CHANGING

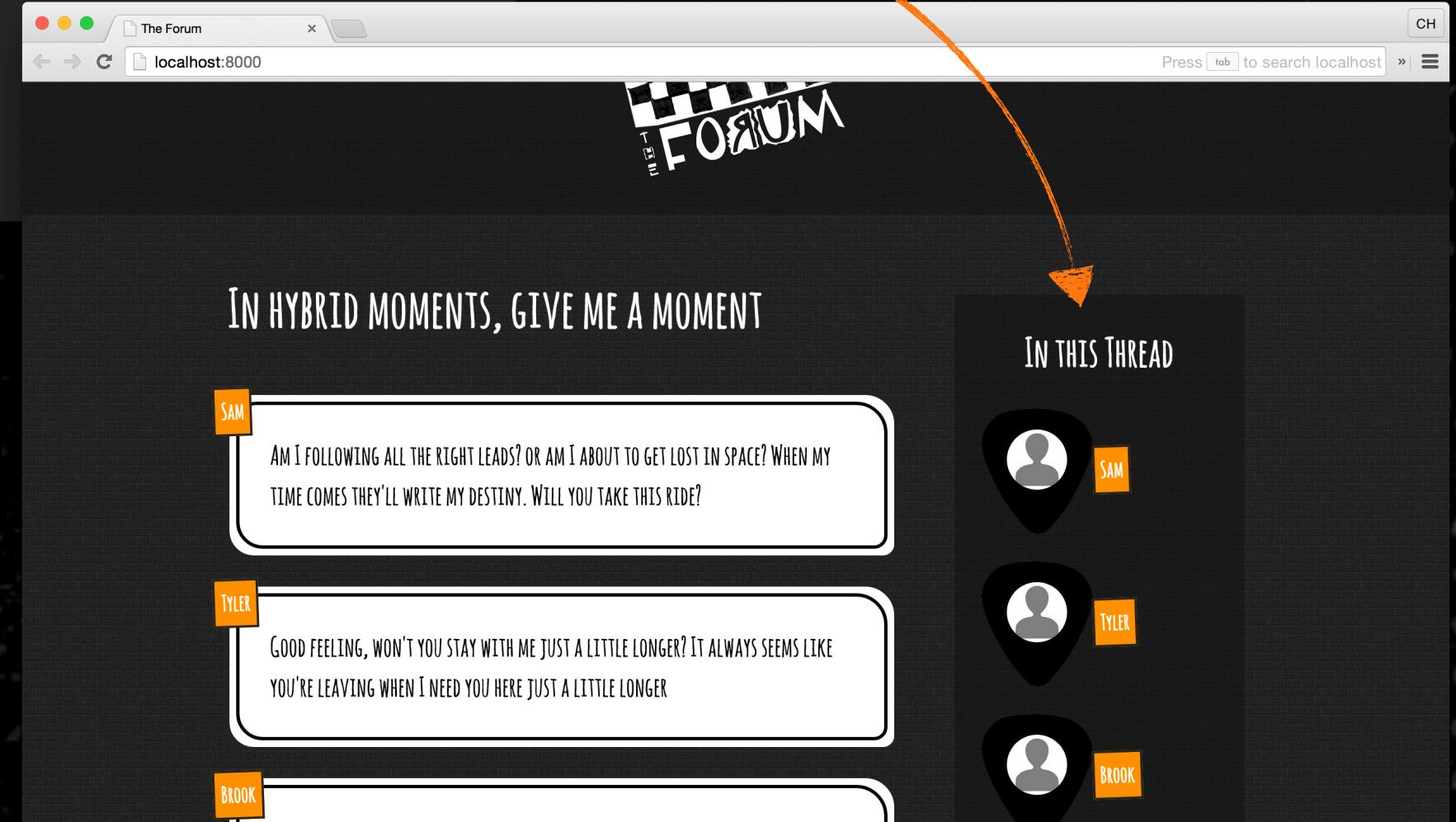
On the right side, there are profile icons for CARLOS S., SAMANTHA A., and JON F., each associated with their respective replies.

Together, we'll build features that are part of a web forum!

Loading Profiles

The `loadProfiles` function takes an array of users and adds their profile to the sidebar.

```
<!DOCTYPE html>
//...
<script src="./load-profiles.js">
<script>
    loadProfiles(["Sam", "Tyler", "Brook"]);
</script>
</body>
</html>
```



The loadProfiles Function

This function displays a different message according to the number of arguments.

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    var loadingMessage = "This might take a while...";  
    _displaySpinner(loadingMessage);  
  }else{  
    var flashMessage = "Loading Profiles";  
    _displayFlash(flashMessage);  
  }  
  //... fetch names and build sidebar  
}
```

Might take a while to process,
so a loading message is displayed

Shouldn't take that long, so
we display a different message

How loadProfiles Is Executed

```
loadProfiles(["Sam", "Tyler", "Brook", "Alex"]);
```

1. Executes the if block

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    var loadingMessage = "This might take a while...";  
    _displaySpinner(loadingMessage);  
  } else{  
    var flashMessage = "Loading Profiles";  
    _displayFlash(flashMessage);  
  }  
  //...  
}
```

2. The loadingMessage variable is declared and assigned a value

3. The displaySpinner function is called

The else block is not executed

Unexpected Behavior With var

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    var loadingMessage = "This might take a while...";  
    _displaySpinner(loadingMessage);  
    console.log(flashMessage); → > undefined  
  } else{  
    var flashMessage = "Loading Profiles";  
    _displayFlash(flashMessage);  
  }  
  console.log(flashMessage); → > undefined  
}
```

flashMessage is never declared...

This indicates a
console output

...but outputs undefined and does not generate error (?!)

Why no ReferenceError ?



Understanding Hoisting

Prior to executing our code, JavaScript moves `var` declarations all the way up to the top of the scope. This is known as **hoisting**.

```
function loadProfiles(userNames){  
  var loadingMessage, flashMessage; // ← Automatically moved here by the JavaScript runtime  
  
  if(userNames.length > 3){  
    var loadingMessage = "This might take a while...";  
    _displaySpinner(loadingMessage);  
  }else{  
    var flashMessage = "Loading Profiles";  
    _displayFlash(flashMessage);  
  }  
  
  //....  
}
```

Declaring Variables With let

let variables are scoped to the nearest block and are NOT hoisted.
(A block is any code section within **curly braces**, like *if*, *else*, *for*, *while*, etc.)

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    let loadingMessage = "This might take a while...";  
    _displaySpinner(loadingMessage);  
  }else{  
    let flashMessage = "Loading Profiles";  
    _displayFlash(flashMessage);  
  }  
  //....  
}
```

Not visible outside if block

Not visible outside else block

Expected Behavior With `let`

Using `let`, variables are “trapped” inside their respective `if` and `else` blocks.

```
loadProfiles(["Sam", "Tyler", "Brook", "Alex"]);
```

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    let loadingMessage = "This might take a while...";  
    _displaySpinner(loadingMessage);  
  }  
  else{  
    let flashMessage = "Loading Profiles";  
    _displayFlash(flashMessage);  
  }  
}
```

```
console.log(flashMessage);
```

→ > ReferenceError: flashMessage is not defined

Expected behavior, similar to
other programming languages!



```
}
```

```
console.log(flashMessage);
```

→ > ReferenceError: flashMessage is not defined

```
}
```

Declarations With *let* *in* *for* Loops

Level 1 - Section 2

Problem With var in for Loops

var is the reason behind a popular “gotcha” in *for* loops.

```
function loadProfiles(userNames){  
//....
```

```
for(var i in userNames){  
  _fetchProfile("/users/" + userNames[i], function(){  
    console.log("Fetched for ", userNames[i]);  
  }  
}  
}
```



Accessing *i* from
inside the callback

```
loadProfiles(["Sam", "Tyler", "Brook", "Alex"]);
```

Unexpectedly outputs the
same value on all iterations



- > Fetched for Alex

Understanding Hoisting and for Loops

var i is hoisted to the top of the function and **shared** across each iteration of the loop.

```
function loadProfiles(userNames){  
  var i;  
  //....  
  
  for( i in userNames){  
    _fetchProfile("/users/" + userNames[i], function(){  
      console.log("Fetched for ", userNames[i]);  
    }  
  }  
}
```

fetchProfile is called 4 times,
before any of the callbacks are invoked

i = 0
i = 1
i = 2
i = 3

i is incremented
on each iteration

Loop Values in Callbacks

When callbacks begin to run, *i* holds the last value assigned to it from the *for* loop.

The diagram illustrates a common pitfall in JavaScript when using callbacks with loops. It shows a `loadProfiles` function that iterates over `userNames` and makes multiple asynchronous `fetchProfile` calls. The variable `i` is used in both the loop and the callbacks, but its value is determined at the time the callback is executed, leading to unexpected results.

Code:

```
function loadProfiles(userNames){  
  var i;  
  //....  
  
  for(i in userNames){  
    _fetchProfile("/users/" + userNames[i], function(){  
      console.log("Fetched for ", userNames[i]);  
    });  
  }  
}  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});
```

Diagram Labels:

- Red X:** A red circle with a white 'X' is located in the top right corner of the slide.
- Prints `userNames[3]` all 4 times:** An orange annotation points to the `console.log` statement in the final `fetchProfile` call, indicating that all four callbacks will print the same value.
- `i = 3`:** Four blue boxes labeled `i = 3` are placed near the `i` variables in the code and the `i` parameters in the `fetchProfile` callbacks, highlighting the variable's value at runtime.

Using let in for Loops

With *let*, there's no sharing in *for* loops. A new variable is created on each iteration.

```
function loadProfiles(userNames){  
  //....  
  
  for(let i in userNames){  
    _fetchProfile("/users/" + userNames[i], function(){  
      console.log("Fetched for ", userNames[i]);  
    });  
  }  
}  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});  
  
fetchProfile("/users/" + userNames[i], function(){  
  console.log("Fetched for ", userNames[i]);  
});
```



Using let in for Loops

Each callback function now holds a reference to their own version of *i*.

```
function loadProfiles(userNames){  
  //....  
  
  for(let i in userNames){  
    _fetchProfile("/users/" + userNames[i], function(){  
      console.log("Fetched for ", userNames[i]);  
    });  
  }  
}
```



```
loadProfiles(["Sam", "Tyler", "Brook", "Alex"]);
```

Outputs the correct
value for each iteration



- > Fetched for Sam
- > Fetched for Tyler
- > Fetched for Brook
- > Fetched for Alex

let Cannot Be Redeclared

Variables declared with `let` can be reassigned, but cannot be redeclared within the same scope.

```
let flashMessage = "Hello";  
flashMessage = "Goodbye";
```



Reassigning is allowed

```
let flashMessage = "Hello";  
let flashMessage = "Goodbye";
```



Redeclaring is not allowed



> `TypeError: Identifier 'name' has already been declared`

```
let flashMessage = "Hello";  
  
function loadProfiles(userNames){  
  let flashMessage = "Loading profiles";  
  return flashMessage;  
}
```



Different scopes

Declarations With *const*

Level 1 – Section 3

Magic Numbers

A magic number is a literal value without a clear meaning.

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    //...  
  }else{  
    //...  
  }  
}
```

The number 3 by itself
doesn't tell us much

Issues With Magic Numbers

When used multiple times, magic numbers introduce **unnecessary duplication**, which can lead to **bad code!**

```
function loadProfiles(userNames){  
  if(userNames.length > 3){  
    //...  
  }else{  
    //...  
  }  
  
  if(someValue > 3){  
    //...  
  }  
}
```



Hard to tell whether both numbers serve the same purpose

Replacing Magic Numbers With Constants

The `const` keyword creates **read-only** named constants.

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



Conveys intent and facilitates maintenance

Constants Are Read-only

Once assigned, constants cannot be assigned a new value.

```
function loadProfiles(userNames){  
  
    const MAX_USERS = 3;  
    MAX_USERS = 10; ← Value does not change  
  
    if(userNames.length > MAX_USERS){  
        //...  
    }else{  
        //...  
    }  
  
    const MAX_REPLIES = 3;  
    if(someElement > MAX_REPLIES){  
        //...  
    }  
}
```



Constants Require an Initial Value

Variables declared with *const* must be assigned an initial value.

```
function loadProfiles(userNames){  
  const MAX_USERS = 3; → > SyntaxError: Unexpected token  
  MAX_USERS = 10;
```

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

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



Constants Are Block Scoped

Variables declared with `const` are scoped to the nearest block.

```
function loadProfiles(userNames){  
  
  const MAX_USERS = 3;  
  
  if(userNames.length > MAX_USERS){  
    const MAX_REPLIES = 15; ← Not visible outside the if block  
    //...  
  }else{  
    //...  
  }  
  
  console.log(MAX_REPLIES); → > ReferenceError, MAX_REPLIES is not defined.  
}
```



let vs. const

In most cases, *let* and *const* will behave very similarly. Consider the semantics when choosing one over the other.

```
let loadingMessage = "This might take a while...";  
let currentAge = 50;  
let totalCost = cost + tax;
```

Use *let* when variables could
be reassigned new values

```
const MAX_USERS = 3;  
const SEPARATOR = "%%";  
const ALLOW_EDIT = false;
```

Use *const* when new variables are
not expected to be reassigned new values

Functions

Level 2 – Section 1

Issues With Flexible Function Arguments

Unexpected arguments might cause errors during function execution.

```
loadProfiles(["Sam", "Tyler", "Brook"]);
```



```
loadProfiles();
```

```
loadProfiles(undefined);
```

> **TypeError: Cannot read property 'length' of undefined**

Breaks when called with no arguments



```
function loadProfiles(userNames){  
  let namesLength = userNames.length;  
  //...  
}
```



Cannot assume userNames
will always be assigned

Manual Argument Checks Don't Scale Well

A common practice is to check for presence of arguments as the very first thing in the function.

```
function loadProfiles(userNames){  
  let names = typeof userNames !== 'undefined' ? userNames : [];  
  let namesLength = names.length;  
  // ...  
}
```



Too verbose and doesn't scale
well for multiple arguments

Using Default Parameter Values

Default parameter values help move default values from the function body to the function signature.

```
function loadProfiles(userNames = []){  
  let namesLength = userNames.length;  
  console.log(namesLength);  
}
```



Uses empty array as default value
when no argument is passed

Does not break when invoked with no arguments

```
loadProfiles();
```



> 0

Nor with explicit undefined as argument

```
loadProfiles(undefined);
```



> 0

The Options Object

The **options object** is a widely used pattern that allows user-defined settings to be passed to a function in the form of properties on an object.

```
setPageThread("New Version out Soon!", {  
  popular: true,  
  expires: 10000,  
  activeClass: "is-page-thread"  
});
```

Options object with 3 properties

```
function setPageThread(name, options = {}){  
  
  let popular = options.popular;  
  let expires = options.expires;  
  let activeClass = options.activeClass;  
  //...  
}
```

Options object is second parameter

Assign from properties
to local variables

Issues With the Options Object

The options object makes it hard to know what options a function accepts.

```
setPageThread("New Version out Soon!", {  
  popular: true,  
  expires: 10000,  
  activeClass: "is-page-thread"  
});
```

```
function setPageThread(name, options = {}){  
  let popular = options.popular;  
  let expires = options.expires;  
  let activeClass = options.activeClass;  
  //...  
}
```



Unclear what options this function expects just by looking at its signature

Boilerplate code

Using Named Parameters

Using **named parameters** for optional settings makes it easier to understand how a function should be invoked.



```
function setPageThread(name, { popular, expires, activeClass }){  
  
    console.log("Name: ", name);  
    console.log("Popular: ", popular);  
    console.log("Expires: ", expires);  
    console.log("Active: ", activeClass);  
}
```

Local variables

Now we know which arguments are available

```
setPageThread("New Version out Soon!", {  
    popular: true,  
    expires: 1000,  
    activeClass: "is-page-thread"  
});
```



```
> Name: New Version out Soon!  
> Popular: true  
> Expires: 1000  
> Active: is-page-thread
```



Omitting Certain Arguments on Call

It's okay to omit some options when invoking a function with named parameters.

```
function setPageThread(name, { popular, expires, activeClass }){  
  
    console.log("Name: ", name);  
    console.log("Popular: ", popular);  
    console.log("Expires: ", expires);  
    console.log("Active: ", activeClass);  
}
```

popular is the only named argument being passed

```
setPageThread("New Version out Soon!", {  
    popular: true  
});
```

> Name: New Version out Soon!
> Popular: true
> Expires: undefined
> Active: undefined

No value assigned to remaining parameters

Don't Omit All Named Arguments on Call

It's NOT okay to omit the options argument altogether when invoking a function with named parameters when no default value is set for them.



```
function setPageThread(name, { popular, expires, activeClass }){  
  
    console.log("Name: ", name);  
    console.log("Popular: ", popular);  
    console.log("Expires: ", expires);  
    console.log("Active: ", activeClass);  
}
```

```
setPageThread("New Version out Soon!");
```

Invoking this function without its second argument breaks our code

↓
> TypeError: Cannot read property 'popular' of undefined



Setting a Default for Named Parameters

Setting a **default value** for the entire options argument allows this parameter to be omitted during function calls.

```
function setPageThread(name, { popular, expires, activeClass } = {}){  
  
    console.log("Name: ", name);  
    console.log("Popular: ", popular);  
    console.log("Expires: ", expires);  
    console.log("Active: ", activeClass);  
}
```



```
setPageThread("New Version out Soon!");
```

We can now safely invoke this function without its second argument



- > Name: New Version out Soon!
- > Popular: undefined
- > Expires: undefined
- > Active: undefined

Rest Parameter, Spread Operator, and Arrow Functions

Level 2 - Section 2

Adding Tags to a Thread

The screenshot shows a web browser window titled "The Forum" at "localhost:8000". The main content area displays a thread with two posts. The first post, by user "BROOK", contains the lyrics: "YOU'RE LEAVING WHEN I NEED YOU HERE JUST A LITTLE LONGER". The second post, also by "BROOK", contains the lyrics: "STATIC PULSE INSIDE OF MUSIC BRINGING US ESCAPE. IT'S ALWAYS TEMPORARY CHANGING NOTHING IN ITS WAKE". To the right of the posts, there is a user profile icon for "BROOK" with the name "BROOK" next to it. Below the posts, there is a section labeled "Tags related to this thread" with an orange arrow pointing to "CATEGORIES: SONGS, LYRICS, BANDS". At the bottom, there is a reply form with a "REPLY" button and a blue "SUBMIT" button.

The Forum

localhost:8000

YOU'RE LEAVING WHEN I NEED YOU HERE JUST A LITTLE LONGER

BROOK

STATIC PULSE INSIDE OF MUSIC BRINGING US ESCAPE. IT'S ALWAYS TEMPORARY CHANGING
NOTHING IN ITS WAKE

BROOK

Tags related to this thread → CATEGORIES: SONGS, LYRICS, BANDS

REPLY

SUBMIT

Functions With a Variable List of Arguments

Here's how we want our *displayTags* function to look.

```
displayTags("songs");
```

or

```
displayTags("songs", "lyrics");
```

or

```
displayTags("songs", "lyrics", "bands");
```

variadic functions can accept
any number of arguments

Issues With the `arguments` Object

The `arguments` object is a built-in, Array-like object that corresponds to the arguments of a function. Here's why relying on this object to read arguments is not ideal:

```
function displayTags(){  
  for(let i in arguments){  
    let tag = arguments[i];  
    _addToTopic(tag);  
  }  
}
```



Hard to tell which parameters
this function expects to be called with

Where did this come from?!

If we add an argument...

```
function displayTags(targetElement){
```

```
  let target = _findElement(targetElement);
```

```
  for(let i in arguments){  
    let tag = arguments[i];  
    _addToTopic(target, tag);  
  }  
}
```



...we'll break the loop, since the
first argument is no longer a tag

Using Rest Parameters

The new **rest parameter** syntax allows us to represent an indefinite number of arguments as an **Array**. This way, changes to function signature are **less likely to break code**.

```
function displayTags(...tags){  
  for(let i in tags){  
    let tag = tags[i];  
    _addToTopic(tag);  
  }  
}
```

These 3 dots are part of the syntax

tags is an Array object

Must always go last

```
function displayTags(targetElement, ...tags){
```

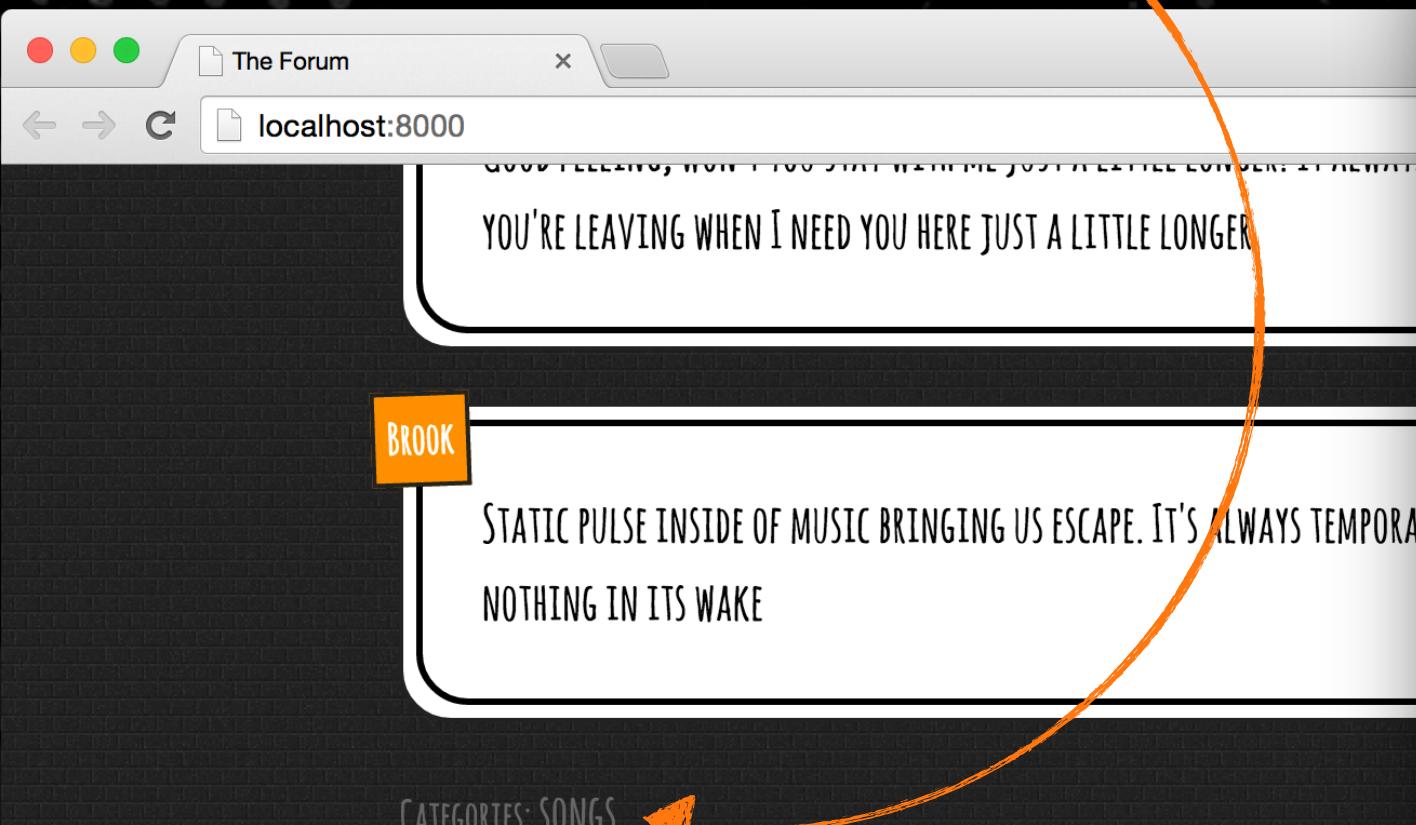
```
let target = _findElement(targetElement);
```

```
for(let i in tags){  
  let tag = tags[i];  
  _addToTopic(target, tag);  
}
```

Not affected by changes
to function signature

Seeing displayTags in Action

displayTags("songs");



displayTags("songs", "lyrics", "bands");



Splitting Arrays Into Individual Arguments

We need a way to convert an **Array** into individual arguments upon a **function call**.

```
getRequest("/topics/17/tags", function(data){  
  let tags = data.tags;  
  displayTags(tags);  
})
```



tags is an Array, e.g., `["programming", "web", "HTML"]` ...

...but `displayTags` expects to be called
with individual arguments, like this:

```
displayTags("programming");
```

```
displayTags("programming", "javascript");
```

*How can we convert Arrays
into individual elements on
a function call ?*

Using the Spread Operator

The spread operator allows us to split an Array argument into individual elements.

```
getRequest("/topics/17/tags", function(data){  
  let tags = data.tags;  
  displayTags(...tags);  
})
```

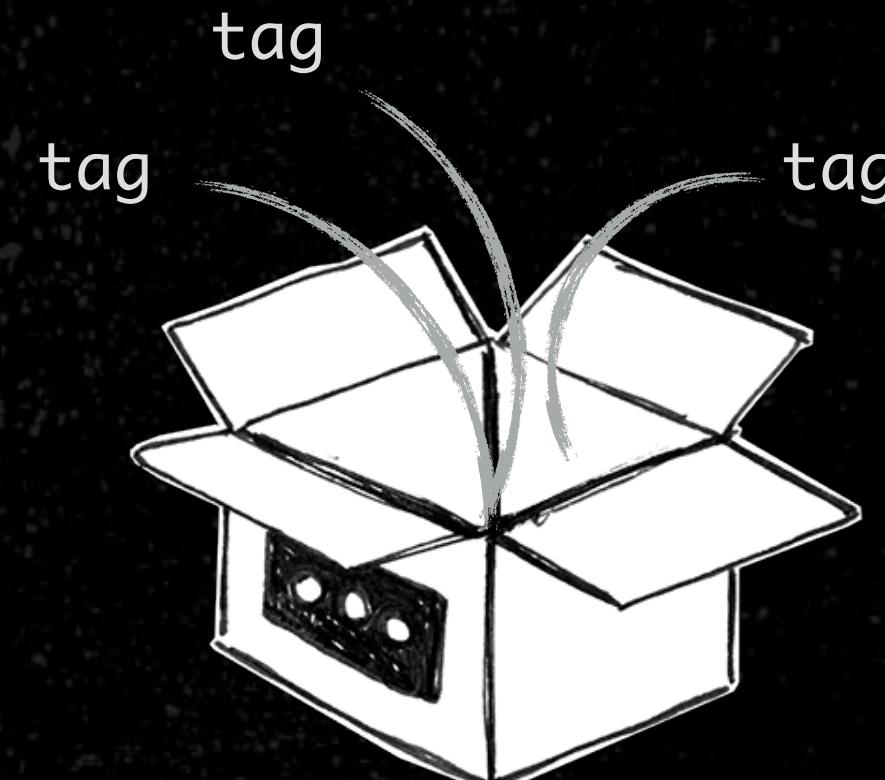


The `displayTags` function is now receiving individual arguments, not an Array

Same as doing this

```
displayTags(tag, tag, tag);
```

Three dots are part of the syntax



Rest and Spread look the same

Rest parameters and the spread operator look the same, but the former is used in **function definitions** and the latter in **function invocations**.

Rest Parameters

```
function displaytags(...tags){  
  for(let i in tags){  
    let tag = tags[i];  
    _addToTopic(target, tag);  
  }  
}
```

Function definition

vs.

Function invocation

Spread Operator

```
getRequest("/topics/17/tags", function(data){  
  let tags = data.tags;  
  displayTags(...tags);  
})
```

From Functions to Objects

JavaScript objects can help us with the encapsulation, organization, and testability of our code.

```
getRequest("/topics/17/tags", function(data){  
  let tags = data.tags;  
  displayTags(...tags);  
})
```

Functions like `getRequest` and `displayTags`
should not be exposed to caller code

We want to convert code like this...

...into code like this

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");  
tagComponent.render();
```

*Let's see how we can implement our **TagComponent** function!*

Creating TagComponent

The *TagComponent* object **encapsulates** the code for fetching tags and adding them to a page.

```
function TagComponent(target, urlPath){  
    this.targetElement = target;  
    this.urlPath      = urlPath;  
}
```

Properties set on the constructor function...

```
TagComponent.prototype.render = function(){  
    getRequest(this.urlPath, function(data){  
        //...  
    });  
}
```

...can be accessed from other instance methods

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");  
tagComponent.render();
```

Passing target element and the URL path as arguments

Issues With Scope in Callback Functions

Anonymous functions passed as callbacks to other functions create their own scope.

```
function TagComponent(target, urlPath){  
  this.targetElement = target;  
  this.urlPath      = urlPath;  
}
```



The scope of the TagComponent object...

```
TagComponent.prototype.render = function(){  
  
  getRequest(this.urlPath, function(data){  
    let tags = data.tags;  
    displayTags(this.targetElement, ...tags);  
  });  
}
```

...is not the same as...

...the scope of the anonymous function

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");  
tagComponent.render();
```

Returns undefined

Using Arrow Functions to Preserve Scope

Arrow functions bind to the scope of where they are defined, not where they are called.

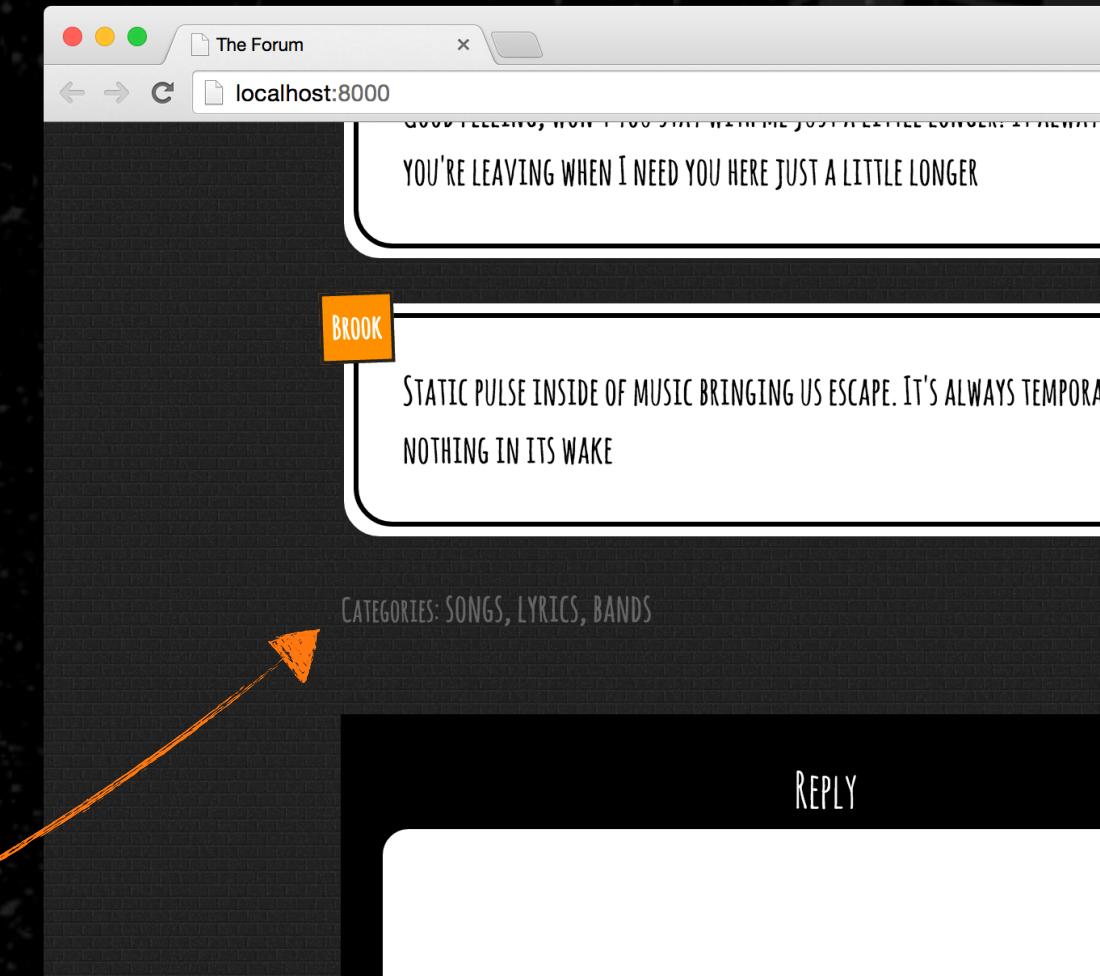
(also known as **lexical binding**)

```
function TagComponent(target, urlPath){
  this.targetElement = target;
  this.urlPath      = urlPath;    Arrow functions bind to the lexical scope
}
```

```
TagComponent.prototype.render = function(){
  getRequest(this.urlPath, (data) => {
    let tags = data.tags;
    displayTags(this.targetElement, ...tags);
  });
}
```

this now properly refers to the TagComponent object

```
let tagComponent = new TagComponent(targetDiv, "/topics/17/tags");
tagComponent.render();
```



Objects and Strings

Level 3 – Section 1

Removing Repetition From Creating Objects

The `buildUser` function returns an object with the `first`, `last`, and `fullName` properties.

```
function buildUser(first, last){  
  let fullName = first + " " + last;  
  
  return { first: first, last: last, fullName: fullName }  
}
```



Returning objects with keys and variables with the same name looks repetitive

Calling the `buildUser` function:

```
let user = buildUser("Sam", "Williams");  
  
console.log( user.first );  
console.log( user.last );  
console.log( user.fullName );
```



```
> Sam  
> Williams  
> Sam Williams
```

The Object Initializer Shorthand

We can remove **duplicate** variable names from object properties when those properties have the **same name** as the variables being assigned to them.

```
function buildUser(first, last){  
  let fullName = first + " " + last;  
  
  return { first, last, fullName };  
}
```



Way cleaner!

Yields the same result:

```
let user = buildUser("Sam", "Williams");  
  
console.log( user.first );  
console.log( user.last );  
console.log( user.fullName );
```



> Sam
> Williams
> Sam Williams

Assigning With Object Initializer Shorthand

The object initializer shorthand works **anywhere** a new object is returned, not just from functions.

```
let name = "Sam";
let age = 45;
let friends = ["Brook", "Tyler"];
```

```
let user = { name, age, friends };
```

```
console.log( user.name );
console.log( user.age );
console.log( user.friends );
```

Same thing →

```
let user = { name: name, age: age, friends: friends };
```

> Sam
> 45
> ["Brook", "Tyler"]

C

Object Destructuring

We can use shorthand to assign properties from objects to local variables with the **same name**.

```
let user = buildUser("Sam", "Williams");
```

```
let first = user.first;  
let last = user.last;  
let fullName = user.fullName;
```

Same names as properties
from return object

```
let { first, last, fullName } = buildUser("Sam", "Williams");
```

```
console.log( first );  
console.log( last );  
console.log( fullName );
```



Unnecessary repetition

This function returns { first, last, fullName }



```
> Sam  
> Williams  
> Sam Williams
```

Destructuring Selected Elements

Not all properties have to be destructured all the time. We can explicitly select the ones we want.

```
let { fullName } = buildUser("Sam", "Williams");
console.log( fullName );
```



> Sam Williams

Only grabbing fullName from the return object

```
let { last, fullName } = buildUser("Sam", "Williams");
console.log( last );
console.log( fullName );
```



> Williams
> Sam Williams

Grabbing last and fullName from the return object

Recap Object Initializer vs. Destructuring

Object Initializer Shorthand Syntax

```
let name = "Sam";
let age = 45;

let user = { name, age };

console.log(user.name);
console.log(user.age);
```

From variables to object properties

Object Destructuring

```
let { first, last, fullName } = buildUser("Sam", "Williams");

console.log(first);
console.log(last);
console.log(fullName);
```

From object properties to variables

Returns { first, last, fullName }

Adding a Function to an Object

In previous versions of JavaScript, adding a function to an object required specifying the **property name** and then the **full function definition** (including the *function* keyword).

```
function buildUser(first, last, postCount){  
  
    let fullName = first + " " + last;  
    const ACTIVE_POST_COUNT = 10;  
  
    return {  
        first,  
        last,  
        fullName,  
        isActive: function(){  
            return postCount >= ACTIVE_POST_COUNT;  
        }  
    }  
}
```



An anonymous function is assigned to an object property

Using the Method Initializer Shorthand

A new shorthand notation is available for adding a method to an object where the keyword *function* is no longer necessary.

```
function buildUser(first, last, postCount){  
  
  let fullName = first + " " + last;  
  const ACTIVE_POST_COUNT = 10;  
  
  return {  
    first,  
    last,  
    fullName,  
    isActive(){  
      return postCount >= ACTIVE_POST_COUNT;  
    }  
  }  
}
```



Less characters and
easier to read!



Template Strings

Template strings are **string literals** allowing embedded expressions. This allows for a much better way to do **string interpolation**.

```
function buildUser(first, last, postCount){  
  
    let fullName = first + " " + last;  
    const ACTIVE_POST_COUNT = 10;  
    //...  
  
}
```



```
function buildUser(first, last, postCount){  
  
    let fullName = `${first} ${last}`;  
    const ACTIVE_POST_COUNT = 10;  
    //...  
  
}
```



Enclosed by back-ticks, NOT single quotes, and code is wrapped inside dollar sign and curly braces

Writing Multi-line Strings

Template strings offer a new — and much better — way to write multi-line strings.

```
let userName = "Sam";
let admin = { fullName: "Alex Williams" };
let veryLongText = `Hi ${userName},
```

this is a very
very

veeeery
long text.

Regards,
\${admin.fullName}
`;

```
console.log(veryLongText);
```

Newline characters are
part of the template string

> Hi Sam,

this is a very
very

veeeery
long text.

Regards,
Alex Williams

Newline characters
are preserved

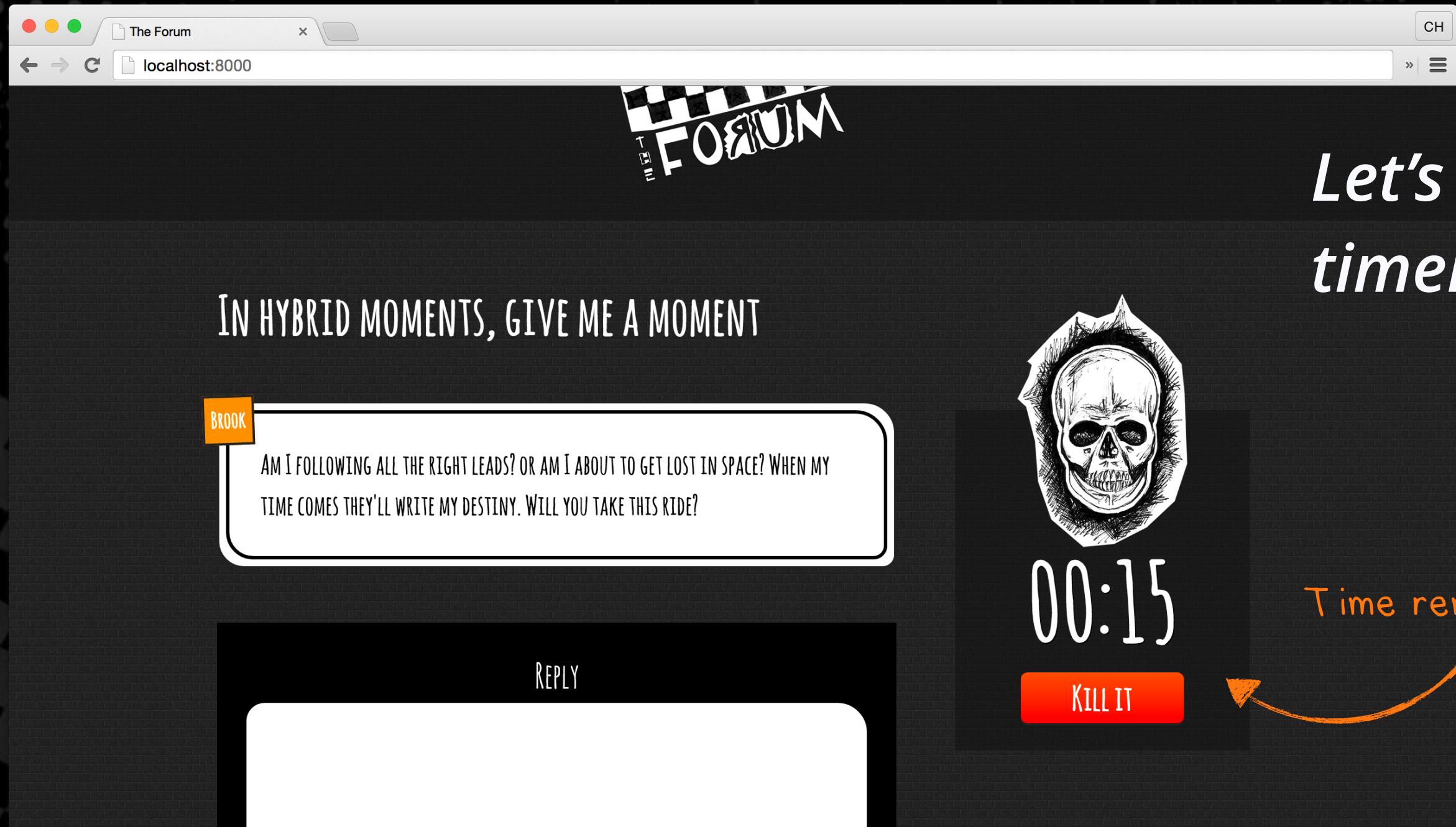


Object.assign

Level 3 – Section 2

Adding a Countdown Timer to Our Forum

The **countdown timer** displays the time left for users to undo their posts after they've been created. Once the time is up, they cannot undo it anymore.



Let's write a countdown timer function!

Writing More Flexible Functions

In order to cater to different applications and domains, our *countdownTimer* function needs to be called in many different ways.

As simple as this...

```
countdownTimer($('.btn-undo'), 60);
```

Two arguments
only

...and as complicated as this

```
countdownTimer($('.btn-undo'), 60, { container: '.new-post-options' });  
  
countdownTimer($('.btn-undo'), 3, { container: '.new-post-options',  
    timeUnit: 'minutes',  
    timeoutClass: '.time-is-up' });
```

Custom options will vary
according to each application

Using Too Many Arguments Is Bad

For functions that need to be used across different applications, it's okay to accept an **options object** instead of using named parameters

```
function countdownTimer(target, timeLeft,  
{ container, timeUnit, clonedDataAttribute,  
  timeoutClass, timeoutSoonClass, timeoutSoonSeconds  
} = {}){  
  
  //...  
}
```

Too many named arguments make
this function harder to read
(Named arguments are okay, but for a different purpose)

```
function countdownTimer(target, timeLeft, options = {}){  
  //...  
}
```



Easier to customize to
different applications

Using Local Values and || Is Bad for Defaults

Some options might not be specified by the caller, so we need to have default values.



```
function countdownTimer(target, timeLeft, options = {}){  
  
  let container = options.container || ".timer-display";  
  let timeUnit = options.timeUnit || "seconds";  
  let clonedDataAttribute = options.clonedDataAttribute || "cloned";  
  let timeoutClass = options.timeoutClass || ".is-timeout";  
  let timeoutSoonClass = options.timeoutSoonClass || ".is-timeout-soon";  
  let timeoutSoonTime = options.timeoutSoonSeconds || 10;  
  
  //...  
}
```



Default strings and numbers
are all over the place... Yikes!

Using a Local Object to Group Defaults

Using a local object to group **default** values for user options is a common practice and can help write more **idiomatic JavaScript**.

```
function countdownTimer(target, timeLeft, options = {}){
```

```
let defaults = {  
  container: ".timer-display",  
  timeUnit: "seconds",  
  clonedDataAttribute: "cloned",  
  timeoutClass: ".is-timeout",  
  timeoutSoonClass: ".is-timeout-soon",  
  timeoutSoonTime: 10  
};
```

```
//...  
}
```



Looks much better

Merging Values Into a Combined Variable

We want to merge *options* and *defaults*. Upon duplicate properties, those from *options* must override properties from *defaults*.

3. ...and the result stored here.

```
let settings
```

```
{  
  container: ...,  
  timeUnit: ...,  
  clonedDataAttribute: ...,  
  timeoutClass: ...,  
  ...  
}
```

=

1. Default properties declared here...

```
let defaults
```

```
{  
  container: ...,  
  timeUnit: ...,  
  clonedDataAttribute: ...,  
  timeoutClass: ...,  
  ...  
}
```

+

2. ...will be merged with these...

```
options
```

```
{  
  timeUnit: ...,  
  timeoutClass: ...,  
}
```

Values passed for *timeUnit* and *timeoutClass*
will override properties on *defaults* object

Merging Values With Object.assign

The `Object.assign` method copies properties from one or more source objects to a target object specified as the very first argument.

```
function countdownTimer(target, timeLeft, options = {}){  
  
  let defaults = {  
    //...  
  };  
  
  let settings = Object.assign({}, defaults, options);  
  //...  
}
```

Merged properties from
defaults and options 

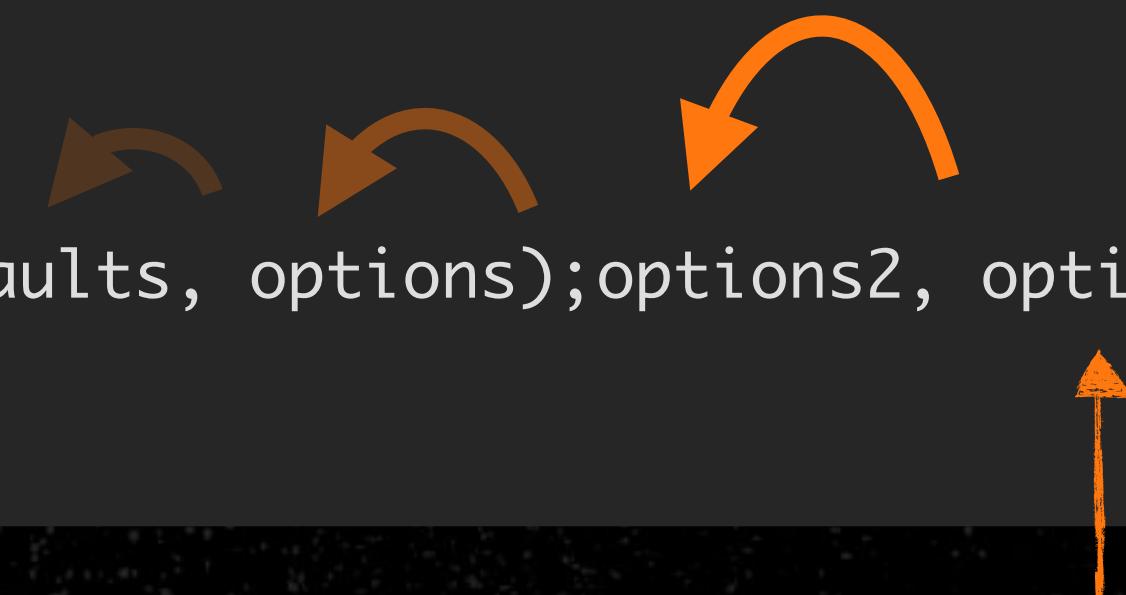
Target object is modified
and used as return value 

Source objects
remain unchanged 

Merging Objects With Duplicate Properties

In case of **duplicate properties** on source objects, the value from the **last object** on the chain always prevails.

```
function countdownTimer(target, timeLeft, options = {}){  
  
  let defaults = {  
    //...  
  };  
  
  let settings = Object.assign({}, defaults, options);options2, options3);  
  //...  
}
```



Duplicate properties from `options3` override those on `options2`, which override those on `options`, etc.

Using Object.assign

There are a couple incorrect ways we might see *Object.assign* being used.

```
Object.assign(defaults, options);
```



defaults is mutated, so we can't go back and access original default values after the merge

```
let settings = Object.assign({}, defaults, options);
```



Can access original default values and looks functional 

```
let settings = {};
Object.assign(settings, defaults, options);
console.log(settings.user);
```



Default values are not changed, but settings is passed as a reference

Not reading return value

Reading Initial Default Values

Preserving the original default values gives us the ability to compare them with the options passed and act accordingly when necessary.

```
function countdownTimer(target, timeLeft, options = {}){  
  
  let defaults = {  
    //...  
  };  
  
  let settings = Object.assign({}, defaults, options);  
  
  if(settings.timeUnit !== defaults.timeUnit){  
    _conversionFunction(timeLeft, settings.timeUnit)  
  }  
}
```



Runs when value passed as argument for timeUnit is different than the original value

Object.assign in Practice

Let's run `countdownTimer()` passing the value for `container` as argument...

```
countdownTimer($('.btn-undo'), 60, { container: '.new-post-options' });
```

...and using the default value for everything else.

```
function countdownTimer(target, timeLeft, options = {}){  
  let defaults = {  
    container: ".timer-display",  
    timeUnit: "seconds",  
    //...  
  };  
  
  let settings = Object.assign({}, defaults, options);  
  
  console.log(settings.container);  
  console.log(settings.timeUnit);  
}
```

C

> .new-post-options
> seconds

Arrays

Level 4 – Section 1

Assigning From Array to Local Variables

We typically access array elements by their index, but doing so for more than just a couple of elements can quickly turn into a **repetitive task**.

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

```
let a = users[0];
let b = users[1];
let c = users[2];
```

```
console.log( a, b, c );
```



This will keep getting longer as we need to extract more elements

Reading Values With Array Destructuring

We can use destructuring to assign multiple values from an array to local variables.

```
let users = ["Sam", "Tyler", "Brook"];
let [a, b, c] = users;
console.log( a, b, c );
```



> Sam Tyler Brook

Still easy to understand AND less code

Values can be discarded

```
let [a, , b] = users;
console.log( a, b );
```



> Sam Brook

Notice the blank space between the commas

Combining Destructuring With Rest Params

We can combine destructuring with rest parameters to group values into other arrays.

```
let users = ["Sam", "Tyler", "Brook"];
let [ first, ...rest ] = users;
console.log( first, rest );
```



> Sam ["Tyler", "Brook"]



Groups remaining arguments in an array

Destructuring Arrays From Return Values

When returning arrays from **functions**, we can assign to multiple **variables** at once.

```
function activeUsers(){  
  let users = ["Sam", "Alex", "Brook"];  
  return users;  
}
```

Returns an array, as expected...

```
let active = activeUsers();  
console.log( active );
```

> ["Sam", "Alex", "Brook"]



...or assigns to **individual variables**. Handy!

```
let [a, b, c] = activeUsers();  
console.log( a, b, c );
```

> Sam Alex Brook



Using *for...of* to Loop Over Arrays

The *for...of* statement iterates over property values, and it's a better way to loop over arrays and other iterable objects.

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

```
for(let index in names){  
  console.log( names[index] );  
}
```

Uses index to read actual element



> Sam Tyler Brook



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

Reads element directly and
with less code involved



> Sam Tyler Brook



Limitations of *for...of* and Objects

The *for...of* statement cannot be used to iterate over properties in plain JavaScript objects out-of-the-box.

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

```
for(let property of post){  
    console.log("Value: ", property);
```



*How do we know when
it's okay to use *for...of*?*

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



Objects That Work With *for...of*

In order to work with *for...of*, objects need a special function assigned to the `Symbol.iterator` property. The presence of this property allows us to know whether an object is **iterable**.

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

```
console.log( typeof names[Symbol.iterator] );
```

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

Symbols are a new data type
guaranteed to be unique



> function

> Sam
> Tyler
> Brook



Since there's a function assigned,
then the names array will work
just fine with *for...of*

Objects That Don't Work With *for...of*

No function assigned to the `Symbol.iterator` property means the object is not iterable.

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

```
console.log( typeof post[Symbol.iterator] );
```



> undefined

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

Nothing assigned to `Symbol.iterator`, so
the post object will not work with `for...of`



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

Finding an Element in an Array

`Array.find` returns the first element in the array that satisfies a provided testing function.

```
let users = [  
  { login: "Sam", admin: false },  
  { login: "Brook", admin: true },  
  { login: "Tyler", admin: true }  
];
```

How can we find an admin in this array of users?

```
let admin = users.find( user => {  
  return user.admin;  
});
```

Returns first object for which `user.admin` is true

```
console.log( admin ); → > { "login": "Brook", "admin": true }
```



One-liner arrow function

```
let admin = users.find( user => user.admin );  
console.log( admin );
```



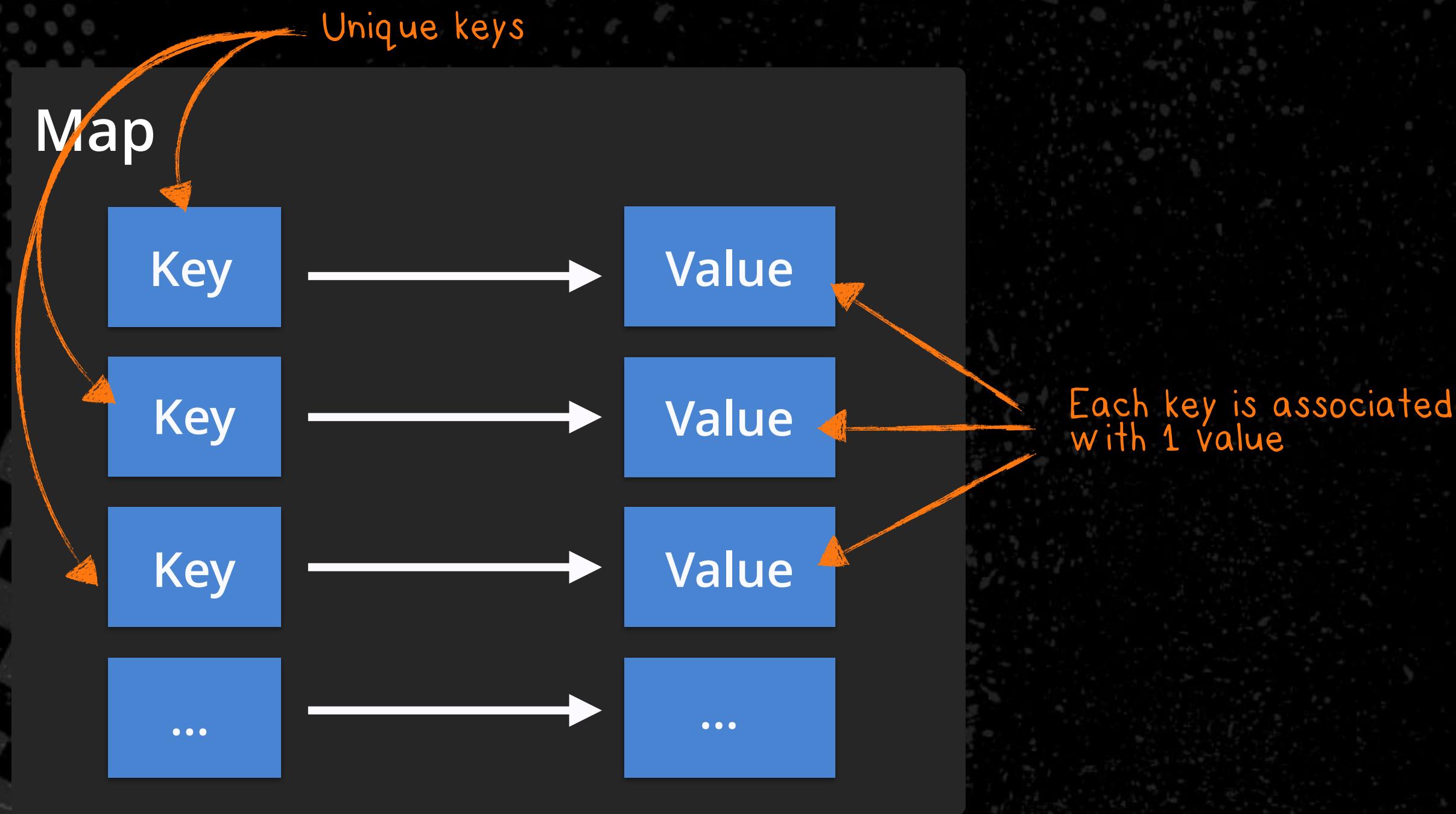
```
> { "login": "Brook", "admin": true }
```

Maps

Level 4 – Section 2

The Map Data Structure

Maps are a **data structure** composed of a collection of **key/value** pairs. They are very useful to store simple data, such as property values.



Issues With Using Objects as Maps

When using Objects as maps, its *keys* are always converted to strings.

Two different objects

```
let user1 = { name: "Sam" };
let user2 = { name: "Tyler" };
```

```
let totalReplies = {};
totalReplies[user1] = 5;
totalReplies[user2] = 42;
```

```
console.log( totalReplies[user1] );
console.log( totalReplies[user2] );
```

```
console.log( Object.keys(totalReplies) );
```



Both objects are converted to
the string "[object Object]"

```
> 42
> 42
```



```
> ["[object Object]"]
```



Storing Key/Values With Map

The `Map` object is a simple **key/value** data structure. Any **value** may be used as either a key or a value, and objects are not converted to strings.

```
let user1 = { name: "Sam" };
let user2 = { name: "Tyler" };
```



```
let totalReplies = new Map();
totalReplies.set( user1, 5 );
totalReplies.set( user2, 42 );
```

Values assigned to different
object keys, as expected

```
console.log( totalReplies.get(user1) );
console.log( totalReplies.get(user2) );
```

> 5
> 42



We use the `get()` and `set()` methods to access values in Maps

Use Maps When Keys Are Unknown Until Runtime

Map

```
let recentPosts = new Map();  
  
createPost(newPost, (data) => {  
  recentPosts.set(data.author, data.message);  
});
```

Keys unknown until runtime, so... Map!

Object

```
const POSTS_PER_PAGE = 15;  
  
let userSettings = {  
  perPage: POSTS_PER_PAGE,  
  showRead: true,  
};
```

Keys are previously defined, so... Object!

Use Maps When Types Are the Same

Map

```
let recentPosts = new Map();  
  
createPost(newPost, (data) => {  
  recentPosts.set( data.author, data.message );  
});  
  
// ...somewhere else in the code  
socket.on('new post', function(data){  
  recentPosts.set( data.author, data.message );  
});
```

All keys are the same type, and
all values are the same type, so Map!

Object

```
const POSTS_PER_PAGE = 15;  
  
let userSettings = {  
  perPage: POSTS_PER_PAGE,  
  showRead: true,  
};
```

Some values are numeric,
others are boolean, so Object!

Iterating Maps With `for...of`

Maps are iterable, so they can be used in a `for...of` loop. Each run of the loop returns a [key, value] pair for an entry in the Map.

```
let mapSettings = new Map();

mapSettings.set( "user", "Sam" );
mapSettings.set( "topic", "ES2015" );
mapSettings.set( "replies", ["Can't wait!", "So Cool"] );
```

```
for(let [key, value] of mapSettings){
  console.log(` ${key} = ${value}`);
}
```

Remember array destructuring ?

> user = Sam
> topic = ES2015
> replies = Can't wait!,So Cool



WeakMap

The *WeakMap* is a type of *Map* where **only objects** can be passed as keys. Primitive data types — such as strings, numbers, booleans, etc. — are **not allowed**.

```
let user = {};  
let comment = {};  
  
let mapSettings = new WeakMap();  
mapSettings.set( user, "user" );  
mapSettings.set( comment, "comment" );  
  
console.log( mapSettings.get(user) );  
console.log( mapSettings.get(comment) );  
  
mapSettings.set("title", "ES2015");
```

> user
> comment



Primitive data types are not allowed

→ > Invalid value used as weak map key

Working With WeakMaps

All available methods on a *WeakMap* require access to an object used as a key.

```
let user = {};  
  
let mapSettings = new WeakMap();  
mapSettings.set( user, "ES2015" );  
  
console.log( mapSettings.get(user) );  
console.log( mapSettings.has(user) );  
console.log( mapSettings.delete(user) );
```

> ES2015
> true
> true



WeakMaps are not iterable, therefore they can't be used with *for...of*

```
for(let [key,value] of mapSettings){  
  console.log(` ${key} = ${value}`);  
}
```

> mapSettings[Symbol.iterator] is not a function



WeakMaps Are Better With Memory

Individual entries in a *WeakMap* can be garbage collected while the *WeakMap* itself still exists.

```
let user = {};
```



All objects occupy memory space

```
let userStatus = new WeakMap();
userStatus.set( user, "logged" );
//...
someOtherFunction( user );
```



Object reference passed as
key to the WeakMap

Once it returns, user can be garbage collected

WeakMaps don't prevent the garbage collector from collecting objects currently used as keys, but that are no longer referenced anywhere else in the system



Sets

Level 4 – Section 3

Limitations With Arrays

Arrays don't enforce uniqueness of items. Duplicate entries are allowed.

```
let tags = [];
```

```
tags.push( "JavaScript" );
tags.push( "Programming" );
tags.push( "Web" );
tags.push( "Web" );
```

Duplicate entry

```
console.log( "Total items ", tags.length );
```

C

> Total items 4

Using Set

The `Set` object stores **unique** values of **any type**, whether primitive values or object references.

```
let tags = new Set();
```

```
tags.add("JavaScript");
tags.add("Programming");
tags.add({ version: "2015" });
tags.add("Web");
tags.add("Web");
```



Both primitive values and objects are allowed

Duplicate entries are ignored

```
console.log("Total items ", tags.size); > Total items 4
```



We use the `add()` method to add elements to a `Set`

Using Set as Enumerable Object

Set objects are **iterable**, which means they can be used with *for...of* and destructuring.

```
let tags = new Set();

tags.add("JavaScript");
tags.add("Programming");
tags.add({ version: "2015" });
tags.add("Web");
```

```
for(let tag of tags){
  console.log(tag);
}
```

```
let [a,b,c,d] = tags;
console.log(a, b, c, d);
```

> JavaScript
> Programming
> { version: '2015' }
> Web

> JavaScript Programming { version: '2015' } Web

Effectively extracting elements via destructuring

WeakSet

The `WeakSet` is a type of `Set` where **only objects** are allowed to be stored.

```
let weakTags = new WeakSet();
```

```
weakTags.add("JavaScript");
weakTags.add({ name: "JavaScript" });
let iOS = { name: "iOS" };
weakTags.add(iOS);
```

```
weakTags.has(iOS);
weakTags.delete(iOS);
```

> `TypeError: Invalid value used in weak set`

Only objects can be added

> `true`
> `true`

WeakSets don't prevent the garbage collector from collecting entries that are no longer used in other parts of the system



Can't Read From a WeakSet

WeakSets cannot be used with *for...of* and they offer no methods for reading values from it.

```
let weakTags = new WeakSet();

weakTags.add({ name: "JavaScript" });
let iOS = { name: "iOS" };
weakTags.add(iOS);

for(let wt of weakTags){
  console.log(wt);
}
```



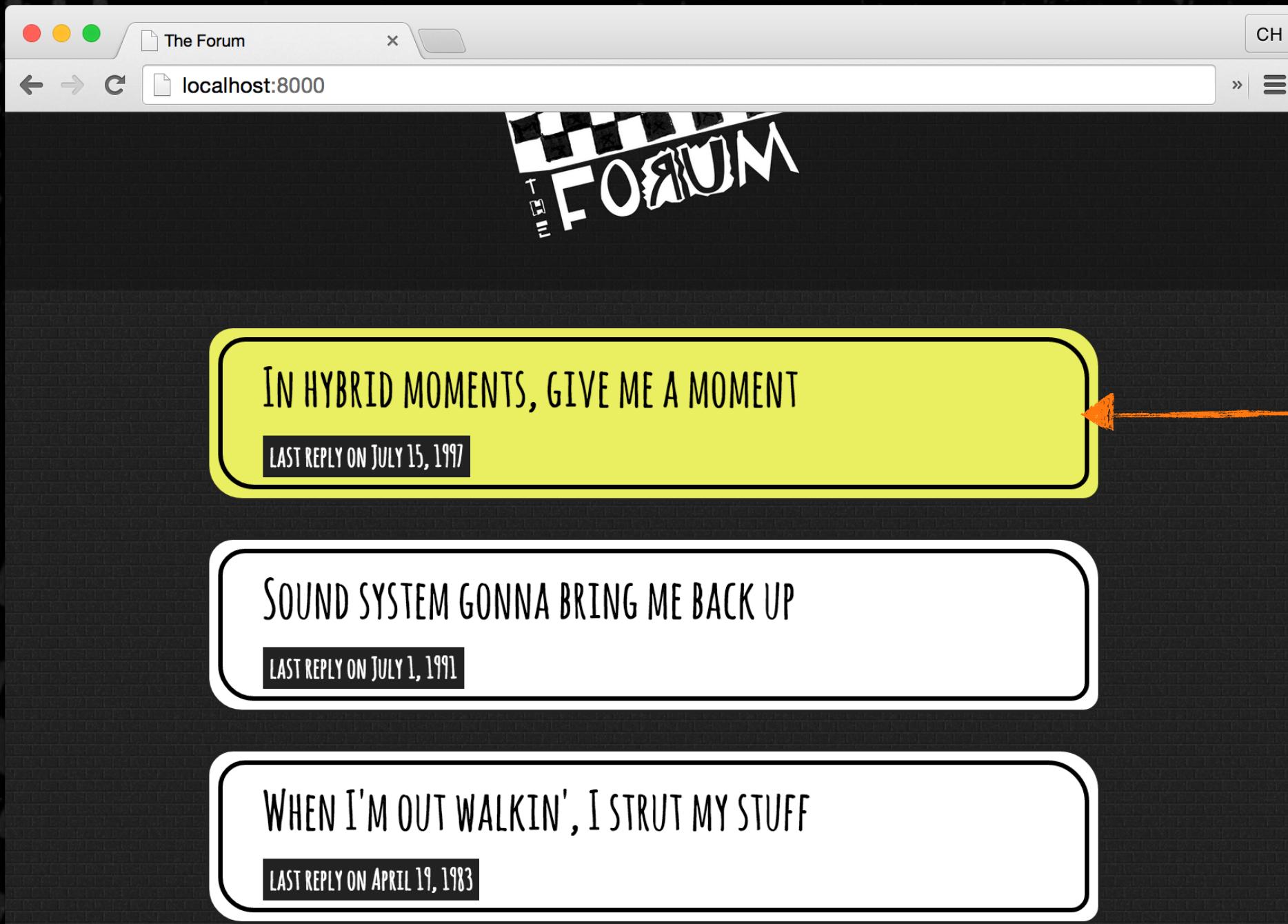
> `TypeError weakTags[Symbol.iterator] is not a function`

Not iterable!

If we can't read values from a WeakSet,
when should we use them ?

Using WeakSets to Show Unread Posts

We want to add a different background color to posts that have not yet been read.



Unread posts should have
a different background color

Using WeakSets to Show Unread Posts

One way to “tag” unread posts is to change a property on each post object once they are read.

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

```
//...when post is clicked on
postList.addEventListener('click', (event) => {
  //...
  post.isRead = true;
});
```

```
// ...rendering list of posts
for(let post of postArray){
  if(!post.isRead){
    _addNewPostClass(post.element);
  }
}
```



Mutates post object in order
to indicate it's been read

Checks a property
on each post object

Showing Unread Posts With WeakSets

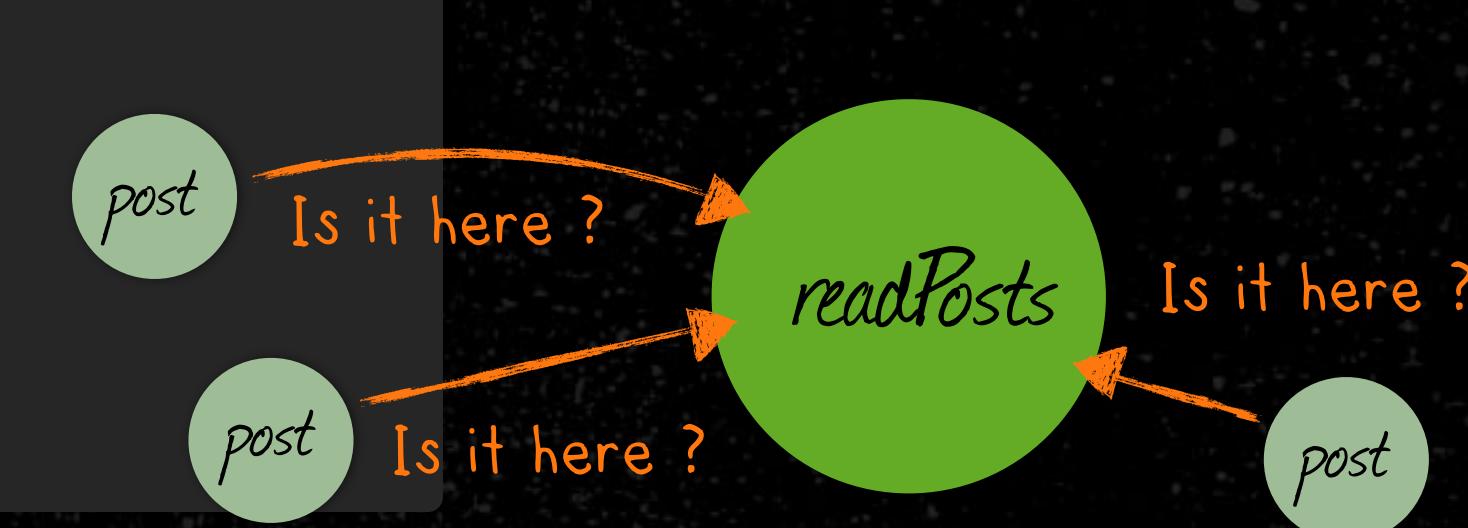
We can use *WeakSets* to create special groups from existing objects without mutating them. Favoring immutable objects allows for much **simpler** code with no **unexpected side effects**.

```
let readPosts = new WeakSet();

//...when post is clicked on
postList.addEventListener('click', (event) => {
    //....
    readPosts.add(post); → Adds object to a group of read posts
});

// ...rendering posts
for(let post of postArray){
    if(!readPosts.has(post)){
        _addNewPostClass(post.element);
    }
}
```

The `has()` method checks whether an object is present in the `WeakSet`

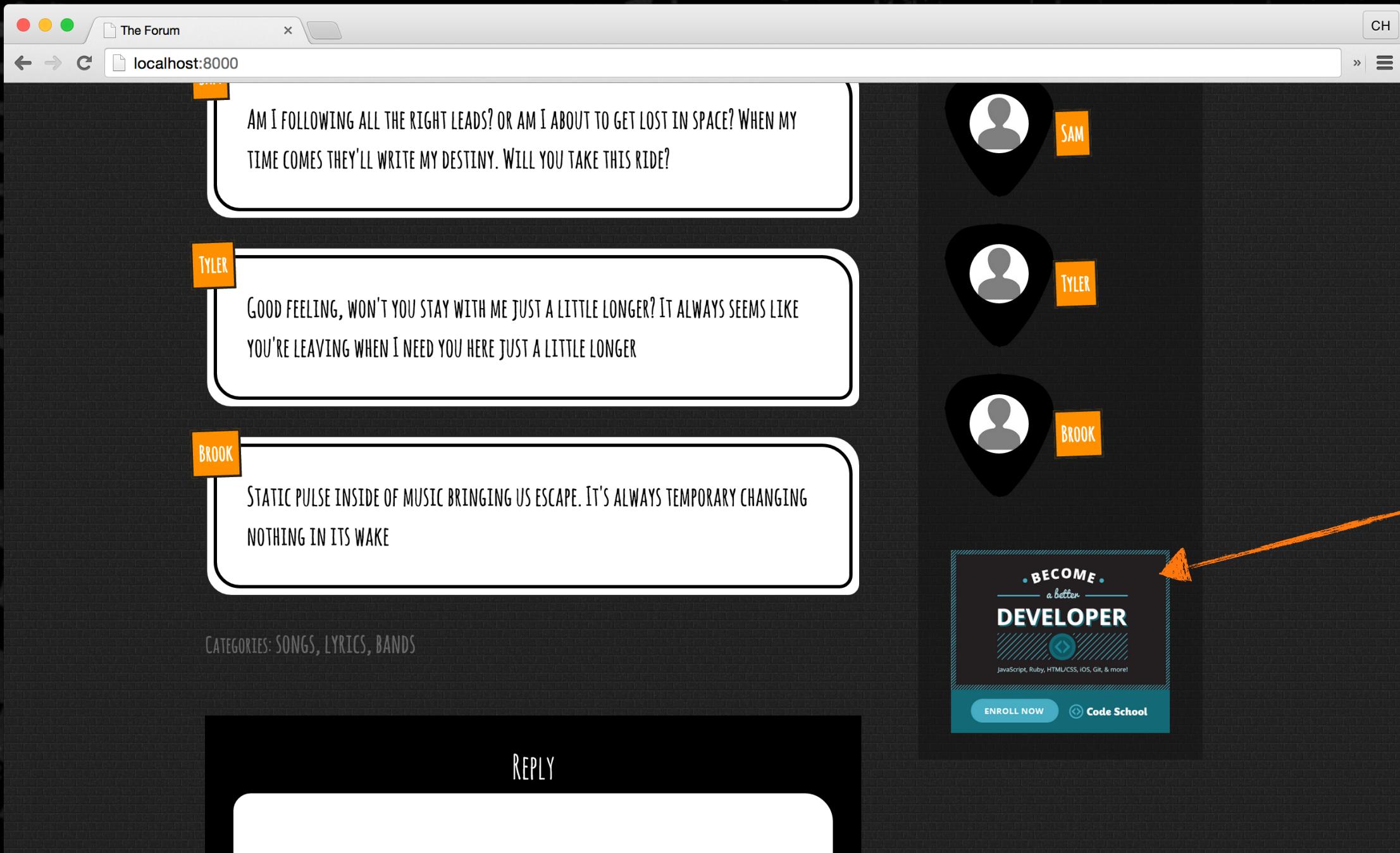


Classes

Level 5 – Section 1

Adding a Sponsor to the Sidebar

We want to add a sponsor widget to the sidebar.



Sponsor widget

Using a Function Approach

A common approach to encapsulation in JavaScript is using a **constructor function**.

```
function SponsorWidget(name, description, url){  
    this.name      = name;  
    this.description = description;  
    this.url       = url;  
}
```

Constructor functions are
invoked with the new operator

```
SponsorWidget.prototype.render = function(){  
    //...  
};
```

Too verbose!

Invoking the *SponsorWidget* function looks like this:

```
let sponsorWidget = new SponsorWidget(name, description, url);  
sponsorWidget.render();
```

Using the New Class Syntax

To define a class, we use the *class* keyword followed by the name of the class. The body of a class is the part between curly braces.

```
class SponsorWidget {  
  render(){ ←  
    //...  
}  
}
```



instance method definitions in classes look just like
the method initializer shorthand in objects!

Initializing Values in the Constructor Function

The `constructor` method is a special method for creating and initializing an object.

```
class SponsorWidget {
```

```
  constructor(name, description, url){
```

```
    this.name      = name;
```

```
    this.description = description;
```

```
    this.url       = url;
```

```
}
```

```
  render(){
```

```
    //...
```

```
}
```

```
}
```

```
let sponsorWidget = new SponsorWidget(name, description, url);  
sponsorWidget.render();
```

Runs every time a new instance is created with the `new` operator

Assigning to instance variables makes them accessible by other instance methods

Still use it just like before

Accessing Class Instance Variables

Instance variables set on the *constructor* method can be accessed from all other instance methods in the class.

```
class SponsorWidget {  
  
  constructor(name, description, url){  
    //...  
    this.url = url;  
  }  
  
  render(){  
    let link = this._buildLink(this.url);  
    //...  
  }  
  
  _buildLink(url){  
    //...  
  }  
}
```

Don't forget to use this to access instance properties and methods

Can access previously assigned instance variables

Prefixing a method with an underscore is a convention for indicating that it should not be invoked from the public API

Creating an Instance From a Class

The class syntax is not introducing a new object model to JavaScript. It's just **syntactical sugar** over the existing **prototype-based inheritance**.

Syntactic Sugar

```
class SponsorWidget {  
  //...  
}
```

Prototype Object Model

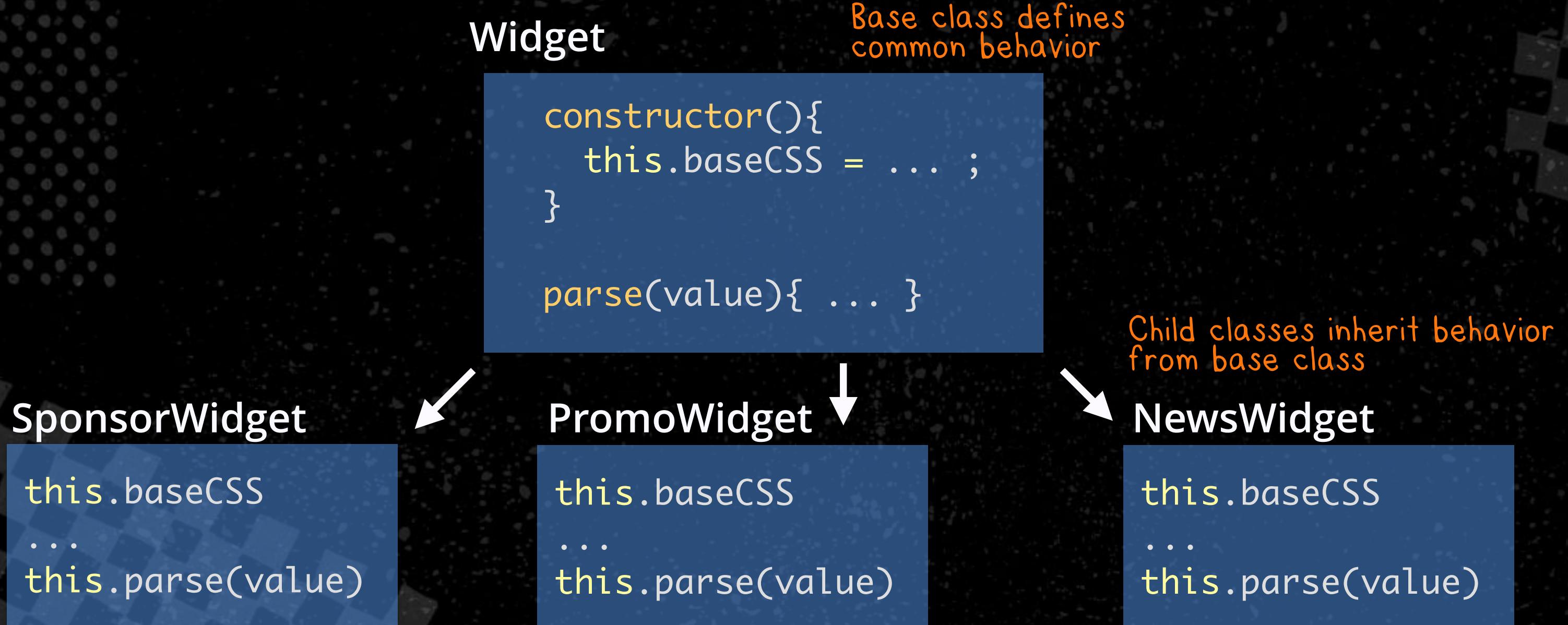
```
function SponsorWidget(name, description, url){  
  //...  
}
```

Instances are created
the same way

```
let sponsorWidget = new SponsorWidget(name, description, url);  
sponsorWidget.render();
```

Class Inheritance

We can use class inheritance to reduce code repetition. Child classes inherit and specialize behavior defined in parent classes.



Using `extends` to Inherit From Base Class

The `extends` keyword is used to create a class that **inherits** methods and properties from another class. The `super` method runs the constructor function from the parent class.

Child Class

Parent Class

```
class Widget {  
  constructor(){  
    this.baseCSS = "site-widget";  
  }  
  
  parse(value){  
    //...  
  }  
}
```

runs parent's setup code

```
class SponsorWidget extends Widget {  
  constructor(name, description, url){  
    super();  
  
    //...  
  }  
  
  render(){  
    let parsedName = this.parse(this.name);  
    let css = this._buildCSS(this.baseCSS);  
    //...  
  }  
}
```

inherits methods

inherits properties

Overriding Inherited Methods

Child classes can invoke methods from their parent classes via the *super* object.

Parent Class

```
class Widget {  
  constructor(){  
    this.baseCSS = "site-widget";  
  }  
  
  parse(value){  
    //...  
  }  
}
```

Child Class

```
class SponsorWidget extends Widget {  
  
  constructor(name, description, url){  
    super();  
    //...  
  }  
  
  parse(){  
    let parsedName = super.parse(this.name);  
    return `Sponsor: ${parsedName}`;  
  }  
  
  render(){  
    //...  
  }  
}
```

Calls the parent version of the *parse()* method

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

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



flash-message.js

flash-message.js

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

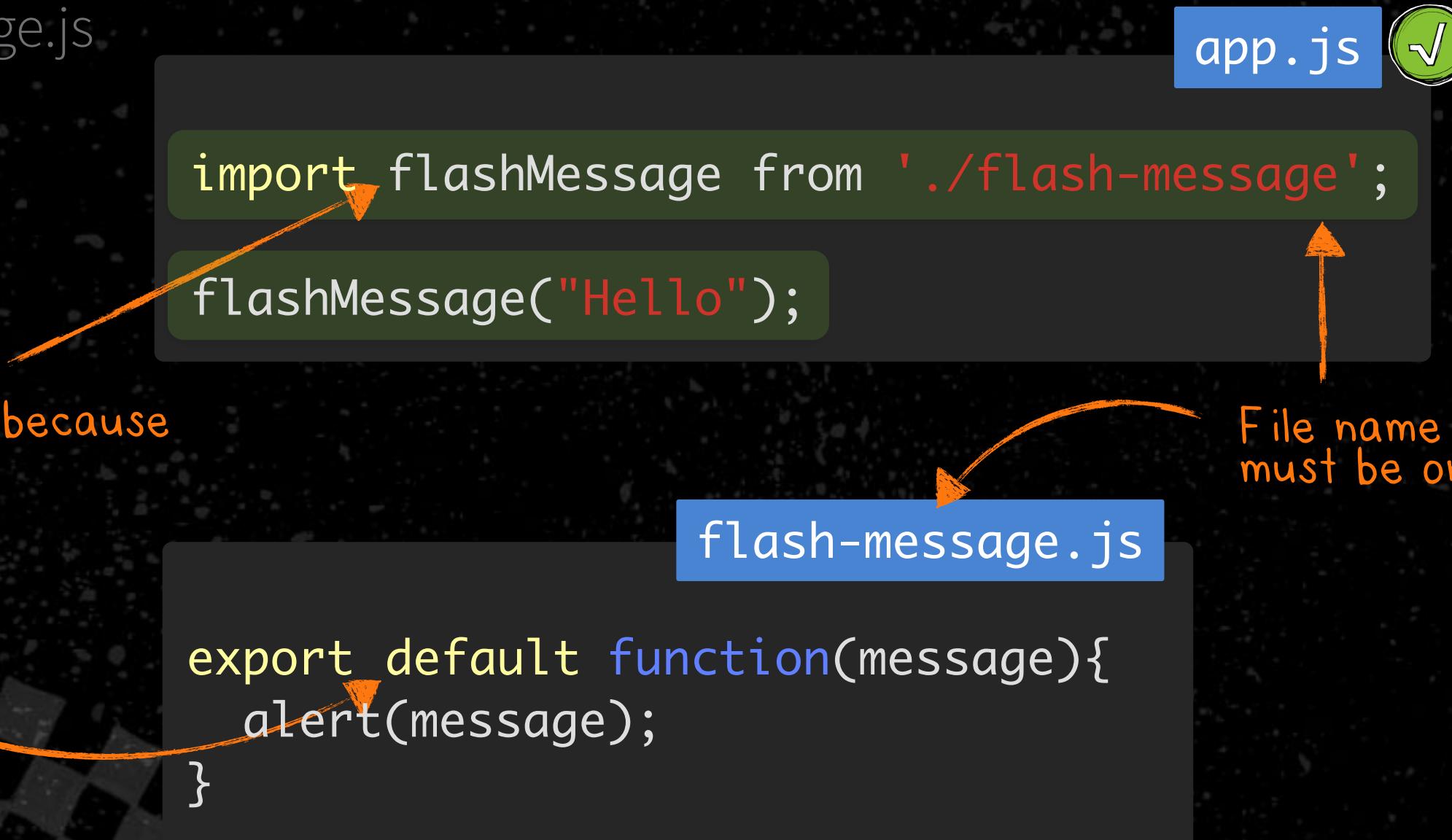
The `export` keyword exposes this
function to the module system

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

Importing Modules With Default Export

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



Running Code From Modules

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

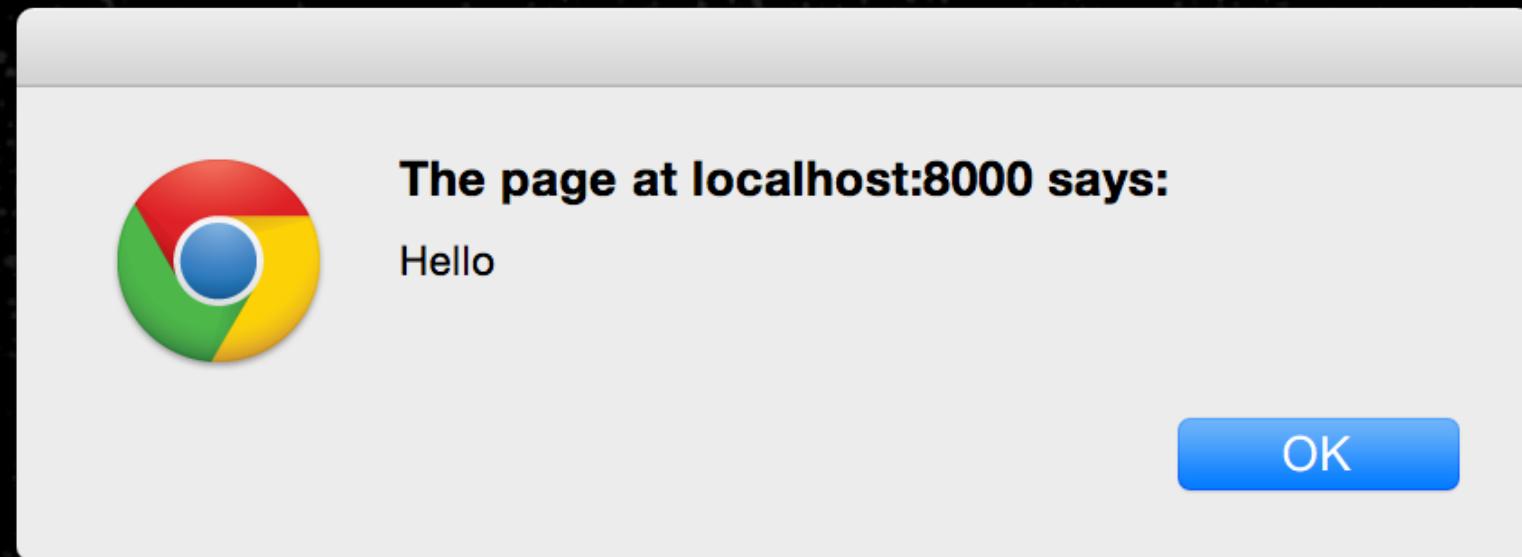
-  flash-message.js
-  app.js
-  index.html

```
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.

app.js

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

flash-message.js

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

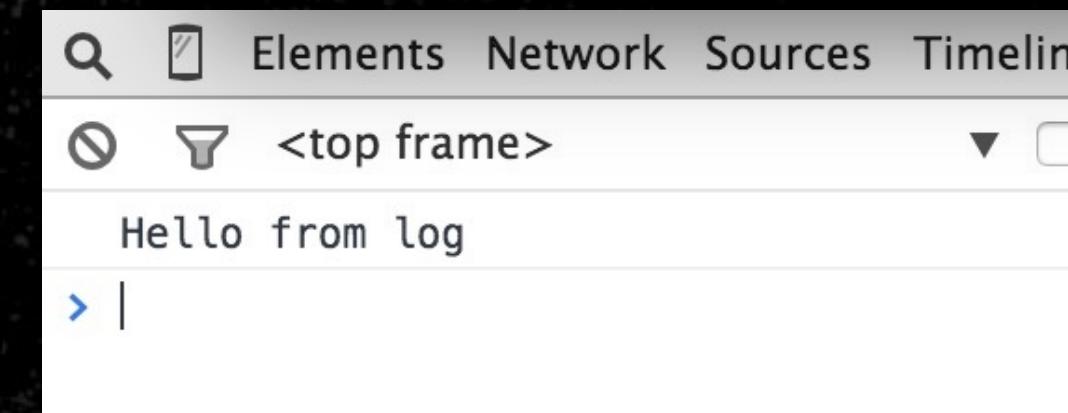
```
import { alertMessage, logMessage } from './flash-message';  
  
alertMessage('Hello from alert');  
logMessage('Hello from log');
```

Names must match



The page at localhost:8000 says:
Hello from alert

OK



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.

app.js

```
import * as flash from './flash-message';
flash.alertMessage('Hello from alert');
flash.logMessage('Hello from log');
```

flash-message.js

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

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

functions become object properties

The page at localhost:8000 says:
Hello from alert

OK

Ele...

<top frame>

Hello from log

OK

Removing Repeated Export Statements

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

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

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

app.js

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

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

flash-message.js

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

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

```
export { alertMessage, logMessage }
```

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 }
```

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.



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

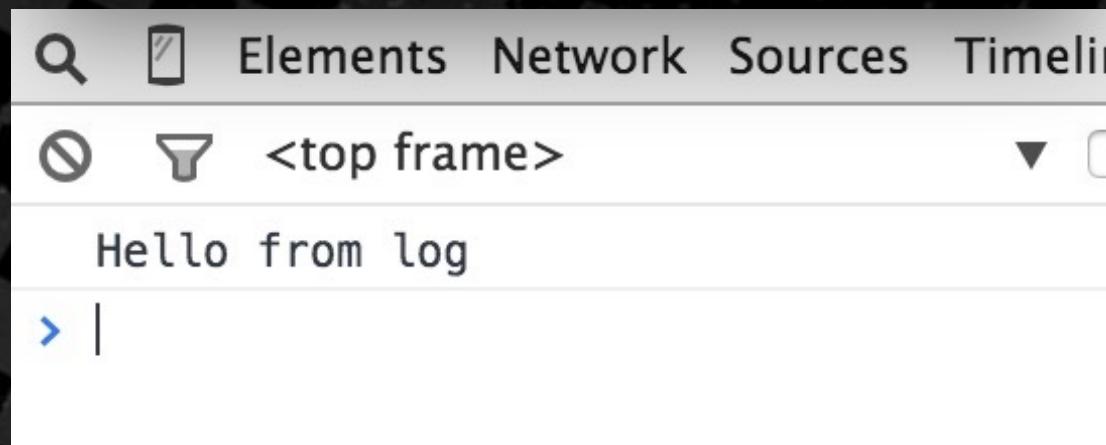
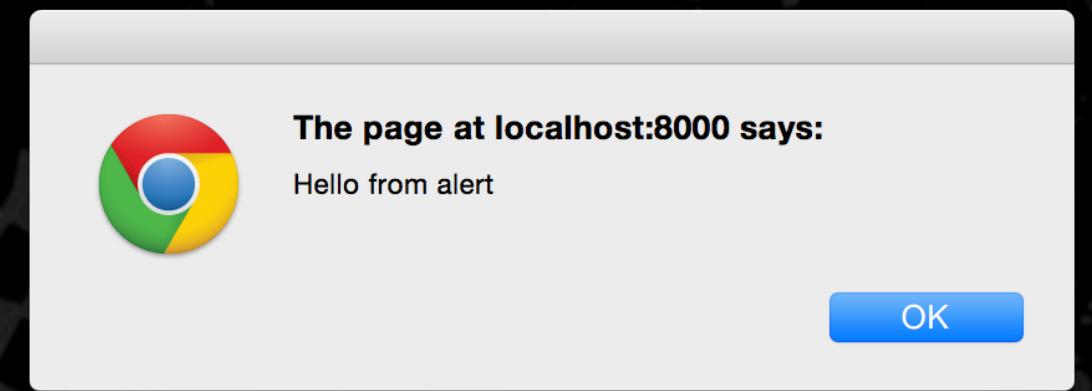
Imported classes are assigned to a variable using *import* and can then be used to create new instances.

- flash-message.js
- app.js

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

flash-message.js

Exporting a class, so F is capitalized



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

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

app.js

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

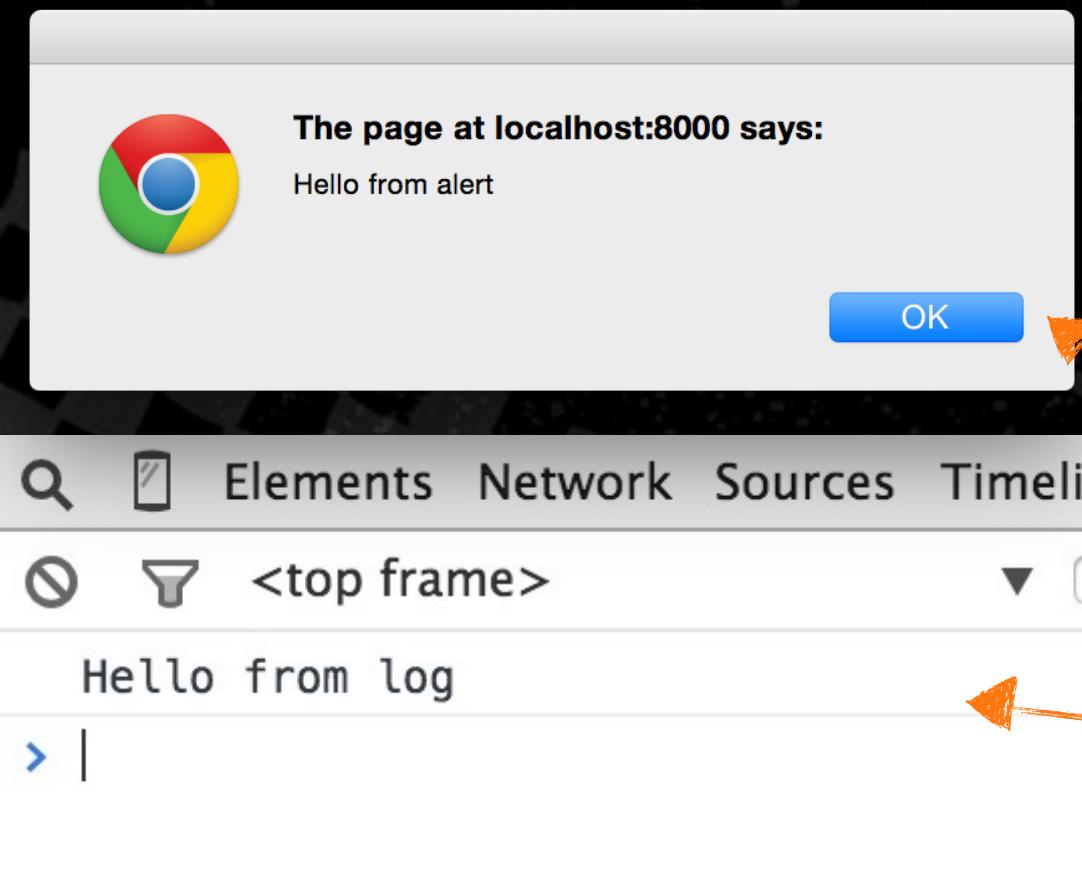
```
class FlashMessage {  
  constructor(message){  
    this.message = message;  
  }  
  
  renderAlert(){  
    alert(`${this.message} from alert`);  
  }  
  
  renderLog(){  
    console.log(`${this.message} from log`);  
  }  
}  
  
export { FlashMessage }
```

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 }  
  
import { FlashMessage } from './flash-message';  
  
let flash = new FlashMessage("Hello");  
flash.renderAlert();  
flash.renderLog();
```

Names must match

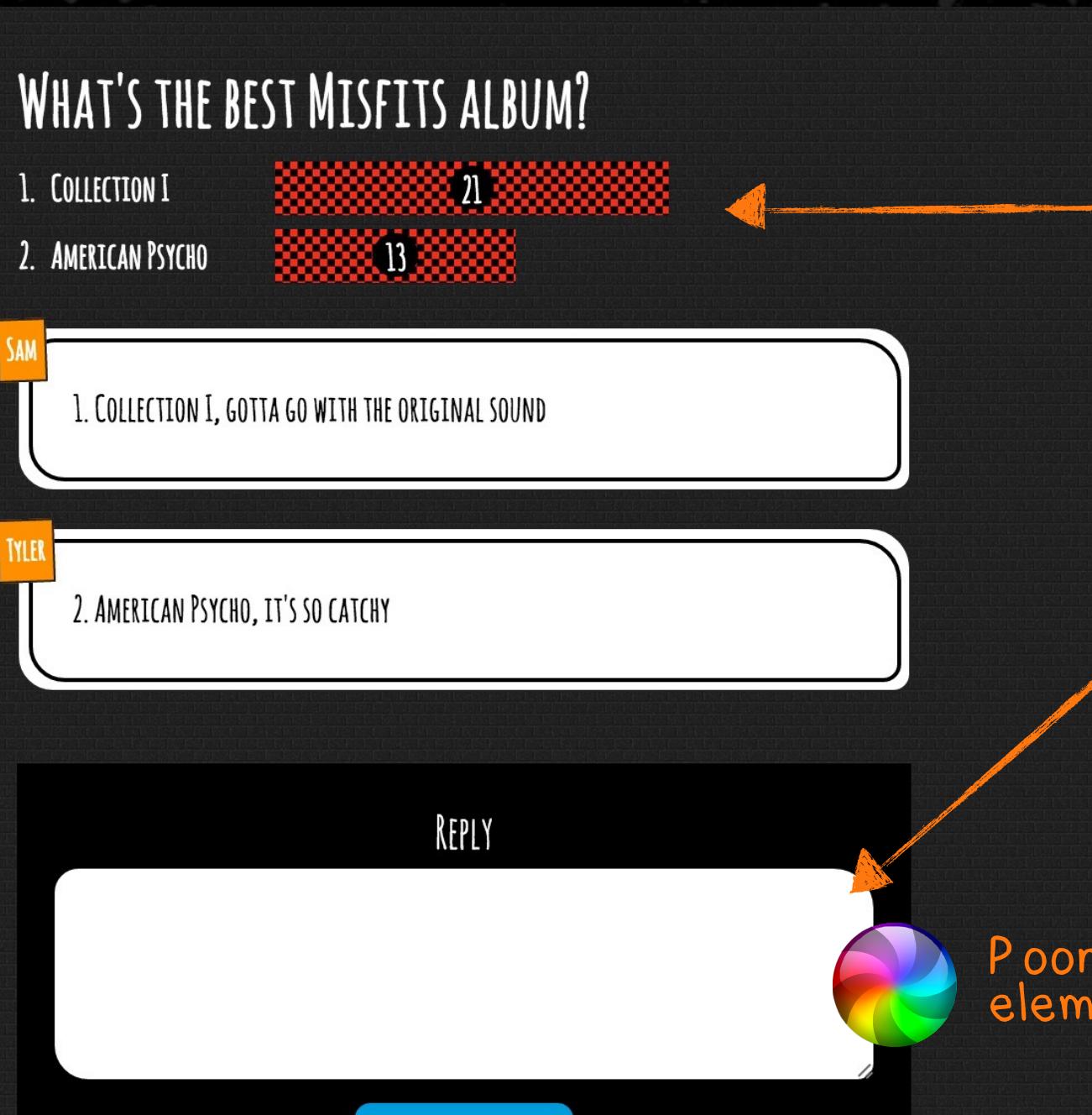
Same result as before

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

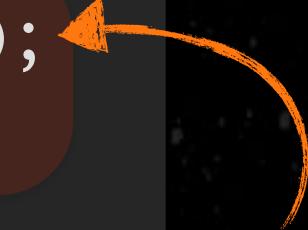
Once the browser **blocks** executing a script, it stops running other scripts, rendering elements, and responding to user events like keyboard and mouse interactions.

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**.



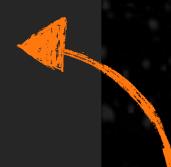
```
getPollResultsFromServer(pollName, function(error, results){  
  if(error){ //.. handle error }  
  //...  
  ui.renderSidebar(results, function(error){  
    if(error){ //.. handle error }  
    //...  
    sendNotificationToServer(pollName, results, function(error, response){  
      if(error){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

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

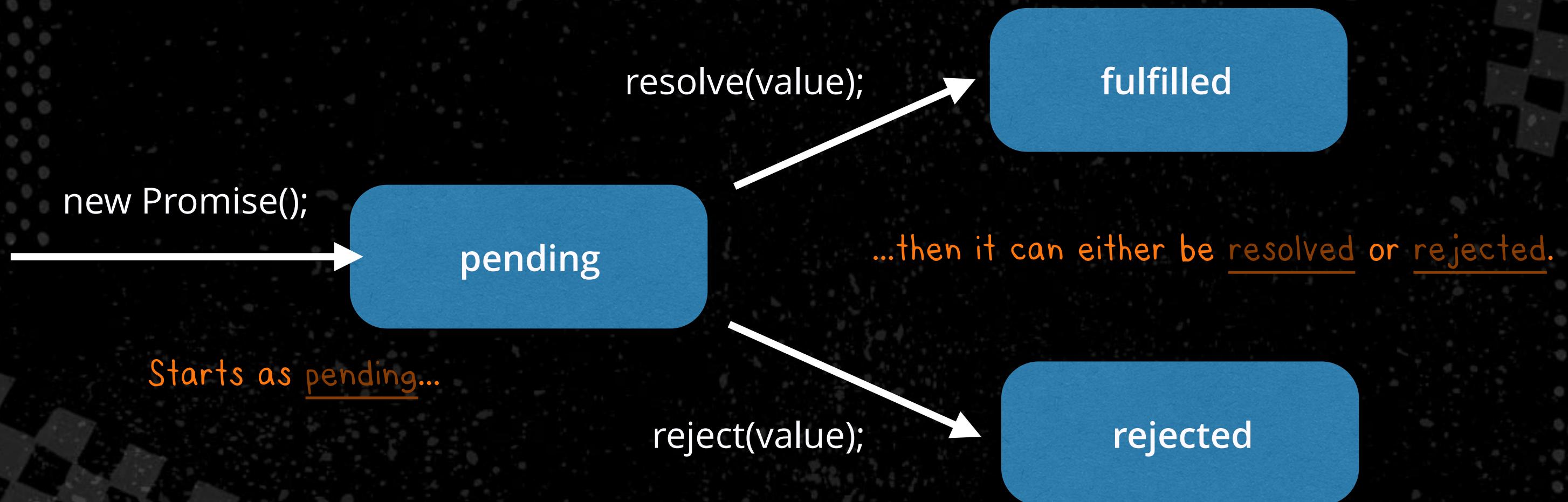
The Promise constructor function takes an anonymous function with 2 callback arguments known as **handlers**.

```
function getPollResultsFromServer(pollName){  
  return new Promise(function(resolve, reject){  
    //...  
    resolve(someValue);           ← Called when the non-blocking  
                                code is done executing  
    //...  
    reject(someValue);          ← Called when an error occurs  
  });  
};
```

Handlers are responsible for either resolving or rejecting the Promise

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

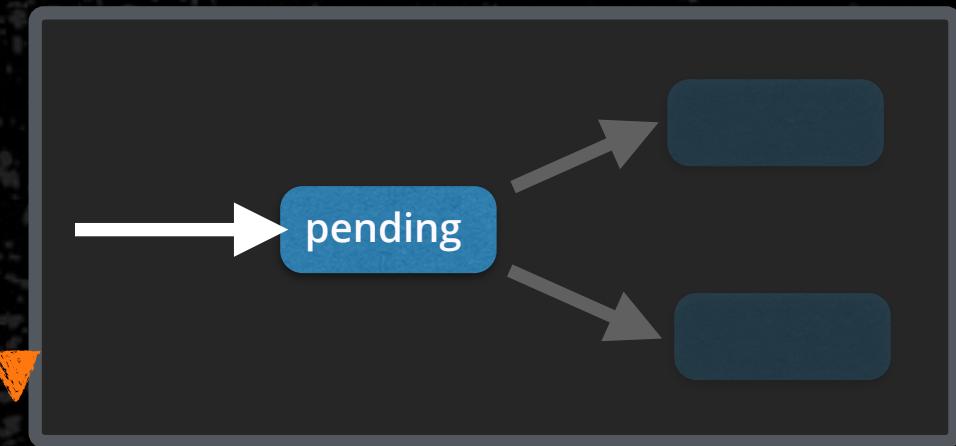
A Promise represents a **future value**, such as the eventual result of an **asynchronous** operation.

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

Not the actual result, but a Promise object

No longer need to pass a callback function as argument

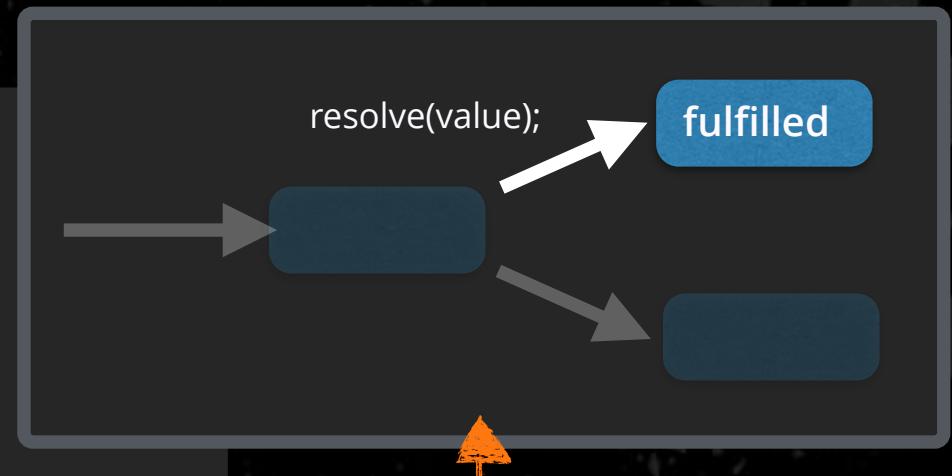
Starts on pending state



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

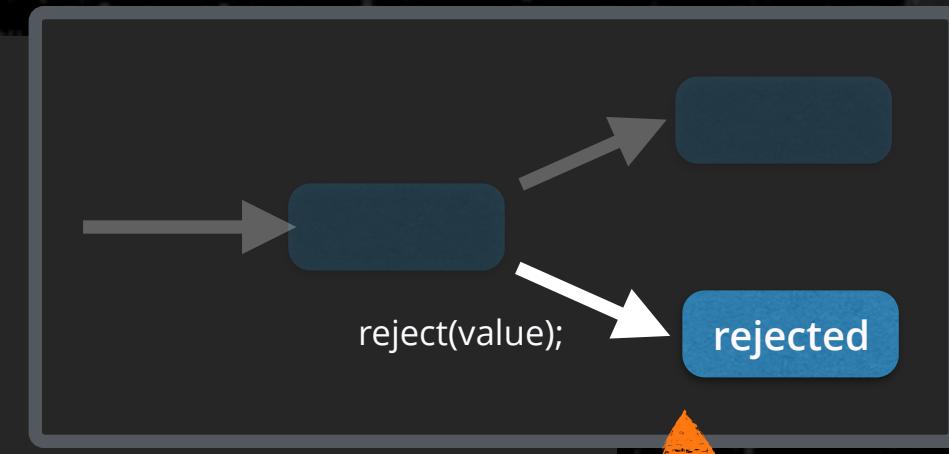
The return value from one call to `then...`

...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){ //... }  
};
```

Remember the new method initializer shorthand syntax?

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

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]();
```

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

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

```
{done: false, value: "Tyler" } ←.....
```

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

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

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

Breaks out of the loop when done is true

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

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

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 };
};

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

>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 };
};
```

The diagram shows the annotated code with several orange annotations:

- An annotation points to the `if(count >= properties.length){` line with the text "Ends the loop after reaching the last property".
- An annotation points to the `this[properties[count++]]` part of the `return` statement with the text "Fetches the value for the next property".
- An annotation points to the `count++` part of the `return` statement with the text "`++` only increments count after it's read".
- An annotation points to the `this` keyword in the `return { next };` line with the text "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

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

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

It's a star character!

```
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"; → { done: false, value: "Sam" }
  yield "Tyler"; → { done: false, value: "Tyler" }
}
```

Calling the function returns a generator object

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

> Sam
> Tyler

C

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

> ["Sam", "Tyler"]

C

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

> Sam Tyler

C

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

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 *(){ ← Generator function signature
```

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