Intermediate JavaScript

Peter J. Jones
pjones@devalot.com
@devalot
http://devalot.com

DEVALOT

Overview

What's In Store

Day 1	Day 2
JavaScript ES2015+	FP part 2
Document Object Model	WebSockets
Functional Programming	Web Storage
OOP and Inheritance	Web APIs
Asynchronous Programming	Developer Tools
Network Calls	Testing

Variable Hoisting

Exercise: Hoisting (Part 1 of 2)

What will the output be?

```
function foo() {
  x = 42;
  var x;

  console.log(x); // ?
  return x;
}
```

Answer: Hoisting (Part 1 of 2)

This:

```
function foo() {
  x = 42;
  var x;

  console.log(x); // ?
  return x;
}
```

Turns into:

```
function foo() {
  var x;
  x = 42;

  console.log(x);
  return x;
}
```

Exercise: Hoisting (Part 2 of 2)

And this one?

```
function foo() {
  console.log(x); // ?
  var x = 42;
}
```

Answer: Hoisting (Part 2 of 2)

This:

```
function foo() {
  console.log(x); // ?
  var x = 42;
}
```

Turns into:

```
function foo() {
  var x;
  console.log(x);
  x = 42;
}
```

Explanation of Hoisting

- Hoisting refers to when a variable declaration is lifted and moved to the top of its scope (only the declaration, not the assignment)
- Function statements are hoisted too, so you can use them before actual declaration
- JavaScript essentially breaks a variable declaration into two statements:

```
var x=0, y;

// Is interpreted as:
var x=undefined, y=undefined;
x=0;
```

Example: Identify the Scope For Each Variable

```
var a = 5;

function foo(b) {
  var c = 10;
  d = 15;

  if (d === c) {
    var e = "error: wrong number";
    console.error(e);
  }

  return function(f) {
    var c = 2;
    return f + c + b;
  };
}
```

Closure Gotcha: Loops, Functions, and Closures

```
// What will this output?
for (var i=0; i<3; i++) {
  setTimeout(function(){
    console.log(i);
  }, 1000*i);
}
console.log("Howdy!");
```

Equality in JavaScript

Sloppy Equality

- The traditional equality operators in JS are sloppy
- That is, they do implicit type conversion

```
"1" == 1;   // true
[3] == "3"; // true

0 != "0";   // false
0 != "";    // false
```

Strict Equality

More traditional equality checking can be done with the === operator:

```
"1" === 1;  // false
0 === "";   // false

"1" !== 1;  // true
[0] !== ""; // true
```

(This operator first appeared in ECMAScript Edition 3, circa 1999.)

Same-Value Equality

Similar to "===" with a few small changes:

```
Object.is(NaN, NaN); // true

Object.is(+0, -0);   // false
```

(This function first appeared in ECMAScript Edition 6, 2015.)

What is the DOM?

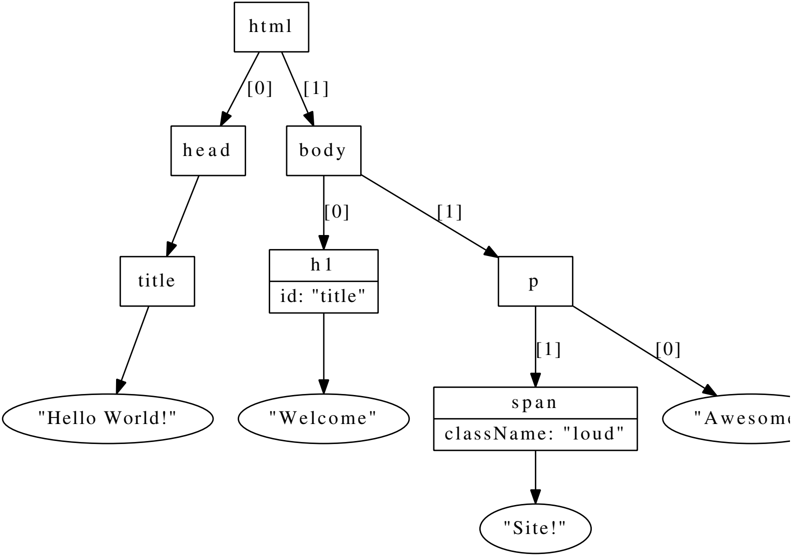
- What most people hate when they say they hate JavaScript
- The DOM is the browser's API for the document
- Through it you can manipulate the document
- Browser parses HTML and builds a tree structure
- It's a live data structure

The Document Structure

- The `document` object provides access to the document
- It's a tree-like structure
- Each node in the tree represents one of:
  - Element
  - Content of an element
- Relationships between nodes allow traversal

Looking at the Parsed HTML Tree (again)

And produce this tree structure:



Element Nodes

- The HTML:

```html
<p id="name" class="hi">My <span>text</span></p>
```

- Maps to:

```javascript
let node = {
  tagName:    "P",
  childNodes: NodeList,
  className:  "hi",
  innerHTML:  "My <span>text</span>",
  id:         "name",
  // ...
};
```

  - Attributes may **very loosely** to object properties

Working with the Document Object Model

- Accessing elements:
  - Select a single element
  - Select many elements
  - Traverse elements
- Working with elements
  - Text nodes
  - Raw HTML
  - Element attributes

Performance Considerations

- Dealing with the DOM brings up a lot of performance issues
- Accessing a node has a cost (especially in IE)
- Styling has a bigger cost (it cascades)
    - Inserting nodes
- Layout changes - Accessing CSS margins - Reflow - Repaint
- Accessing a `NodeList` has a cost

Getting References to Elements

Accessing Individual Elements

Starting on the `document` object or a previously selected element:

`document.getElementById("main");`
: Returns the element with the given ID (e.g., `<div id="main">`).

`document.querySelector("p span");`
: Returns the *first* element that matches the given CSS selector. The search is done using depth-first pre-order traversal.

Accessing a List of Elements

Starting on the `document` object or a previously selected element:

`document.getElementsByTagName("a");` Returns a `NodeList` containing *all* `<a>` elements.

`document.getElementsByClassName("highlight");` Returns a `NodeList` containing *all* elements that have a `class` attribute set to `foo` (e.g., `<div class="highlight">`).

`document.querySelectorAll("p span");` Returns a `NodeList` containing *all* elements that match the given CSS selector.

Traversing the DOM

Traversal Functions

`parentNode` The parent of the specified element.

`previousSibling` The element immediately preceding the specified element.

`nextSibling` The element immediately following the specified element.

`firstChild` The first child element of the specified element.

`lastChild`: The last child element of the specified element.

`childNodes` A `NodeList` containing the direct decedents (children) of the specified element.

But. . .

DOM Living Standard (WHATWG)

Supported in IE $>=$ 9:

`children`: All *element* children of a node (i.e. no text nodes).

`firstElementChild`: First *element* child.

`lastElementChild`: Last *element* child.

`childElementCount`: The number of children that are *elements*.

`previousElementSibling`: The previous sibling that is an *element*.

`nextElementSibling`: The next sibling that is an *element*.

Node Types

The `nodeType` Property

Interesting values for the `element.nodeType` property:

Value	Description
1	Element node
3	Text node
8	Comment node
9	Document node

Manipulating the DOM Tree

Creating New Nodes

`document.createElement("a");` Creates and returns a new node without inserting it into the DOM. In this example, a new <a> element is created.

`document.createTextNode("hello");` Creates and returns a new text node with the given content.

Adding Nodes to the Tree

```
let parent = document.getElementById("customers"),
    existingChild = parent.firstElementChild,
    newChild = document.createElement("li");
```

`parent.appendChild(newChild);` Appends `newChild` to the end of `parent.childNodes`.

`parent.insertBefore(newChild, existingChild);` Inserts `newChild` in `parent.childNodes` just before the existing child node `existingChild`.

`parent.replaceChild(newChild, existingChild);` Removes `existingChild` from `parent.childNodes` and inserts `newChild` in its place.

`parent.removeChild(existingChild);` Removes `existingChild` from `parent.childNodes`.

Node Attributes

Getting and Setting Node Attributes

```
let element = document.getElementById("foo"),
    name    = "bar";
```

`element.getAttribute(name);` Returns the value of the given attribute.

`element.setAttribute(name, value);` Changes the value of the given attribute name to `value`.

`element.hasAttribute(name);` Returns `true` if `element` has an attribute with the given name.

`element.removeAttribute(name);` Removes the named attribute from `element`.

The Class Attribute

Class Attribute API

```
let element = document.getElementById("foo"),
    name    = "bar";
```

`element.classList.add(name);` Add `name` to the list of classes in the class attribute.

`element.classList.remove(name);` Remove `name` from the list of classes in the class attribute.

`element.classList.toggle(name);` If `name` is present in the class list, remove it. Otherwise add it to the class list.

`element.classList.contains(name);` Check to see if the class list contains `name`.

Node Content

HTML and Text Content

```
let element = document.getElementById("foo"),
    name    = "bar";
```

`element.innerHTML` Get or set the element's decedents as HTML.

`element.textContent`: Get or set *all* of the text nodes (including decedents) as a single string.

`element.nodeValue` If `element` is a text node, comment, or attribute node, returns the content of the node.

`element.value` If `element` is a form input, returns its value.

DOM Nodes: Exercises

Exercise: DOM Manipulation

1. Open the following files in your text editor:
   - `src/www/js/flags/flags.js`
   - `src/www/js/flags/index.html` (read only!)
2. Open the `index.html` file in your web browser.
3. Complete the exercise.

Event Handling and Callbacks

Events Overview

- Single-threaded, but asynchronous event model
- Events fire and trigger registered handler functions
- Events can be click, page ready, focus, submit (form), etc.

So Many Events!

- UI: load, unload, error, resize, scroll
- Keyboard: keydown, keyup, keypress
- Mouse: click, dblclick, mousedown, mouseup, mousemove
- Touch: touchstart, touchend, touchcancel, touchleave, touchmove
- Focus: focus, blur
- Form: input, change, submit, reset, select, cut, copy, paste

Using Events (the Basics)

1. Select the element you want to monitor
2. Register to receive the events you are interested in
3. Define a function that will be called when events are fired

Event Registration

Use the `addEventListener` function to register a function to be called when an event is triggered:

Example: Registering a click handler:

```
let main = document.getElementById("main");

main.addEventListener("click", function(event) {
  console.log("event triggered on: ", event.target);
});
```

Note: Don't use older event handler APIs such as `onClick`!

Event Handler Call Context

- Functions are called in the context of the DOM element
- I.e., `this === eventElement`
- Use `bind` or the `let self = this;` trick

Event Propagation

- By default, events propagate from the target node upwards until the root node is reached (bubbling).
- Event handlers can stop propagation using the `event.stopPropagation` function.
- Event handlers can also stop the browser from performing the default action for an event by calling the `event.preventDefault` function

Example: Event Handler

```
main.addEventListener("click", function(event) {
  event.stopPropagation();
  event.preventDefault();

  // ...
});
```

Event Delegation

- Parent receives event instead of child (via bubbling)
- Children can change without messing with event registration
- Fewer handlers registered, fewer callbacks
- Relies on some event object properties:
  - `event.target`: The element the event triggered for
  - `event.currentTarget`: Registered element (parent)

Event Handling: A Complete Example

```
node.addEventListener("click", function(event) {
  // `this' === Node the handler was registered on.
  console.log(this);

  // `event.target' === Node that triggered the event.
  console.log(event.target);

  // Add a CSS class:
  event.target.classList.add("was-clicked");

  // You can stop default browser behavior:
  event.preventDefault();
});
```

Exercise: Simple User Interaction

1. Open the following files in your text editor:
   - `src/www/js/events/events.js`
   - `src/www/js/events/index.html` (read only!)
2. Open the `index.html` file in your web browser.
3. Complete the exercise.

Event Loop Warnings

- Avoid blocking functions (e.g., `alert`, `confirm`)
- For long tasks use iteration or web workers
- Iteration: Break work up using `setTimeout(0)`

Event "Debouncing"

- Respond to events in intervals instead of in real-time
- Reuse a timeout object to process events in the future

```
let input   = document.getElementById("search"),
    output  = document.getElementById("output"),
    timeout = null;

let updateSearchResults = function() {
  output.textContent = input.value;
};

input.addEventListener("keydown", function(e) {
  if (timeout) clearTimeout(timeout);
  timeout = setTimeout(updateSearchResults, 100);
});
```

Defining and Invoking Functions

Defining a Function

There are several ways of defining functions:

- Function statements (named functions)
- Function expression (anonymous functions)
- Arrow functions (new in ES2015)

Function Definition (Statement)

```javascript
function add(a, b) {
  return a + b;
}

let result = add(1, 2); // 3
```

- This syntax is know as a *function definition statement*. It is only allowed where statements are allowed.
- In modern JavaScript you will mostly use the expression form of function definitions or the arrow function syntax.

Function Definition (Expression)

```
let add = function(a, b) {
  return a + b;
};

let result = add(1, 2); // 3
```

- Function is callable through a variable
- Name after `function` is optional
- We'll see it used later

Function Definition (Arrow Functions)

Short form (single expression, implicit `return`):

```
let add = (a, b) => a + b;
add(1, 2);
```

Long form (multiple expressions, explicit `return`):

```
let add = (a, b) => {
  return a + b;
};

add(1, 2);
```

Function Invocation

- Parentheses are mandatory in JavaScript for function invocation
- Any number of arguments can be passed, regardless of the number defined
- Extra arguments won't be bound to a name
- Missing arguments will be `undefined`

Function Invocation (Example)

```javascript
let add = function(a, b) {
  return a + b;
};

add(1)        // a is 1, b is undefined
add(1, 2)     // a is 1, b is 2
add(1, 2, 3) // No name for 3.
```

(Note: ES2015 has default parameters.)

Function Parameters

Special Function Variables

Functions have access to two special variables:

- `arguments`: An object that encapsulates all function arguments
- `this`: The object the function was called through

Rules for Using the `arguments` Variable

- Access all arguments, even unnamed ones
- Array-like, but not an actual array
- Only has `length` property
- Should be treated as read-only (never modify!)
- To treat like an array, convert it to one
- Best to just use ES2015 *rest* parameters

```
let args = Array.prototype.slice.call(arguments);
```

or, with ES2015:

```
let args = Array.from(arguments);
```

Function Arity

A function's *arity* is the number of arguments it expects. In JavaScript you can access a function's arity with its `length` property:

```
function foo(x, y, z) { /* ... */ }
foo.length; // => 3
```

Default Parameters

```
let add = function(x, y=1) {
  return x + y;
};

add(2); // 3
```

- Parameters can have *default* values
- When a parameter isn't bound by an argument it takes on the default value, or `undefined` if no default is set
- Default parameters are evaluated at *call time*
- May refer to any other variables in scope

Rest Parameters

```
let last = function(x, y, ...args) {
  return args.length;
};

last(1, 2, 3, 4); // 2
```

- When an argument name is prefixed with "`...`" it will be an array containing all of the arguments that are not bound to names
- Unlike `arguments`, the rest parameter only contains arguments that are not bound to names
- Unlike `arguments`, the rest parameter is a real `Array`

Spread Syntax

```
let max = function(x, y) {
  return x > y ? x : y;
};

let ns = [42, 99];

max(...ns); // 99
```

- When the name of an array is prefixed with "`...`" in an expression that expects arguments or elements, the array is expanded
- Works when calling functions and creating array literals
- Can be used to splice arrays together

(Object spreading is part of ES2018.)

Function Objects

Functions as Data

Functions can be treated like any other type of JavaScript value:

```
let add = function(a, b) {return a + b;};

let x = add;        // x is now a function object
x(1, 2);            // Same as add(1, 2);
```

Passing Functions as Arguments

It's very common to create functions *on the fly* and pass them to other functions as arguments:

```
let a = [1, 2, 3];

a.forEach(function(n) {
  console.log(n);
});
```

Functions that Return Functions

Functions can create *nested functions* and return them:

```
function recordStartTime() {
  let d = new Date();

  return function() {
    return d;
  };
};

let getStartTime = recordStartTime();
getStartTime(); // 2018-07-03T23:16:00.383Z
```

(Note: this creates what's known as a *closure*.)

Closures

Closures: Basics

- One of the most important features of JavaScript
- And often one of the most misunderstood & feared features
- But, they are all around you in JavaScript
- Happens automatically when you nest functions

Closures: Definitions

- Bound variable: local variables created with `var` or `let` are said to be *bound*.
- Free variable: Any variable that isn't bound and isn't a global variable is called a *free* variable.
- A function that uses free variables *closes around* them, capturing them in a *closure*.
- A closure is a new scope for free variables.

Demonstrating Closures: An Example

```
let makeCounter = function(startingValue) {
  let n = startingValue;

  return function() {
    return n += 1;
  };
};

let counter = makeCounter(0);
counter(); // 1
counter(); // 2
```

(Open `src/examples/js/closure.html` and play in the debugger.)

A Practical Example of Using Closures: Private Variables

Using closures to create truly private variables in JavaScript:

```
let Foo = function() {
  let privateVar = 42;

  return {
    getPrivateVar: function() {
      return privateVar;
    },
    setPrivateVar: function(n) {
      if (n) privateVar = n;
    }
  };
};

let x = Foo();
x.getPrivateVar(); // 42
```

Exercise: Sharing Scope

1. Open the following file:
   src/www/js/closure/closure.js
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Closure Gotcha: Loops, Functions, and Closures

```
// What will this output?
for (var i=0; i<3; i++) {
  setTimeout(function(){
    console.log(i);
  }, 1000*i);
}
console.log("Howdy!");
```

Receivers and Messages

Calling Functions Through Objects

```
let apple  = {name: "Apple",  color: "red"   };
let orange = {name: "Orange", color: "orange"};

let logColor = function() {
  console.log(this.color);
};

apple.logColor  = logColor;
orange.logColor = logColor;

apple.logColor();
orange.logColor();
```

Function.prototype.call

Calling a function and explicitly setting this:

```
let x = {color: "red"};
let f = function() {console.log(this.color);};

f.call(x);          // this.color === "red"
f.call(x, 1, 2, 3); // `this' + arguments.
```

`Function.prototype.apply`

The `apply` method is similar to `call` except that additional arguments are given with an array:

```
let x = {color: "red"};
let f = function() {console.log(this.color);};

f.apply(x); // this.color === "red"

let args = [1, 2, 3];
f.apply(x, args); // `this' + arguments.
```

Function.prototype.bind

The `bind` method creates a new function which ensures your original function is always invoked with `this` set as you desire, as well as any arguments you want to supply:

```
let x = {color: "red"};
let f = function() {console.log(this.color);};

x.f = f;

let g = f.bind(x);
let h = f.bind(x, 1, 2, 3);

g(); // Same as x.f();
h(); // Same as x.f(1, 2, 3);
```

Modules

Modules, Namespaces, and Packages

- Organize logical units of functionality
- Prevent namespace clutter and collisions
- Several options for module implementation
    - The module pattern
    - CommonJS modules
    - ES2015 modules

Immediately-Invoked Function Expressions: Basics

The module pattern:

```
(function() {
  let x = 1;
  return x;
})();
```

Example: Module Pattern

```
let Car = (function() {
  // Private variable.
  let speed = 0;

  // Private method.
  let setSpeed = function(x) {
    if (x >= 0 && x < 100) {speed = x;}
  };

  // Return the public interface.
  return {
    stop: function() {setSpeed(0);},
    inc:  function() {setSpeed(speed + 10);},
  };
})();
```

Exercise: Using IIFEs to Make Private Functions

1. Open the following file:
   src/www/js/hosts/hosts.js
2. Follow the instructions inside the file
3. Open the `index.html` file for the tests

Defining ES2015 Modules

```javascript
const magicNumber = 42;

function sayMagicNumber() {
  console.log(magicNumber);
}

export { sayMagicNumber };
```

Using ES2015 Modules

```
import sayMagicNumber from './module.js';
sayMagicNumber();
```

ES2015 Module Notes

- Not very practical on the client (browser)
- Best as part of the development process:
  - Flattened using a tool such as webpack

Functional Programming with Arrays

Introducing Higher-order Functions

The `forEach` function is a good example of a *higer-order* function:

```
let a = [1, 2, 3];

a.forEach(function(val, index, array) {
  // Do something...
});
```

Or, less idiomatic:

```
let f = function(val) { /* ... */ };
a.forEach(f);
```

Array Testing

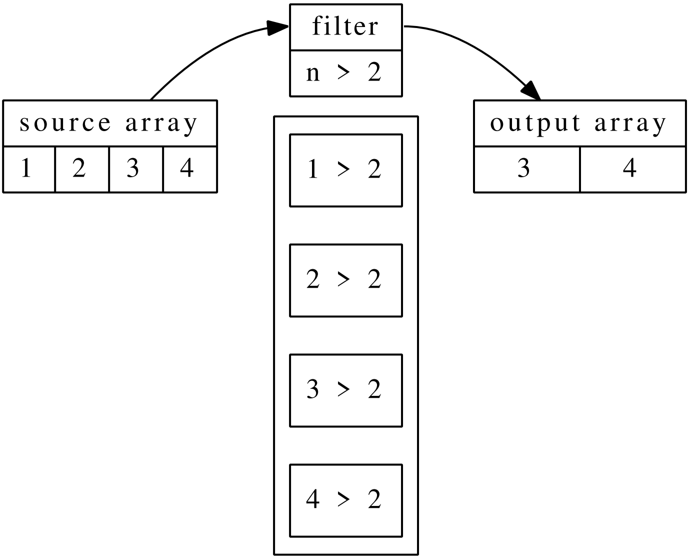
- Test if a function returns `true` on all elements:

```
let a = [1, 2, 3];

a.every(function(val) {
  return val > 0;
});
```

- Test if a function returns `true` at least once:

```
a.some(function(val) {
  return val > 2;
});
```

Filtering an Array with a Predicate Function

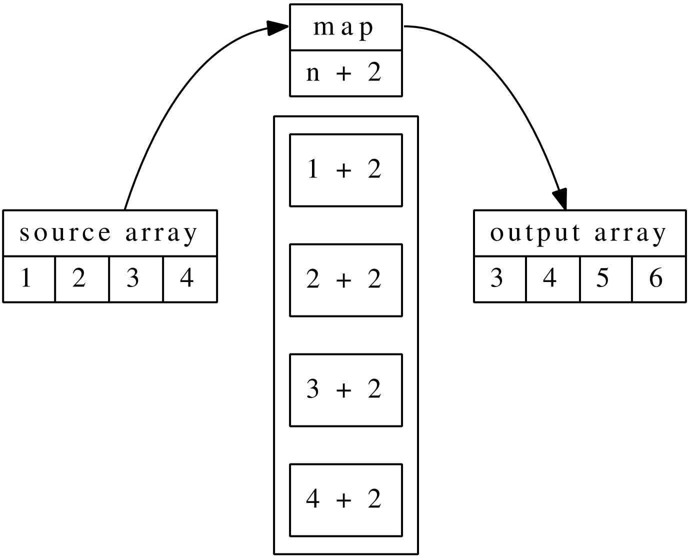


Filter Example

```
let numbers = [10, 7, 23, 42, 95];

let even = numbers.filter(function(n) {
  return n % 2 === 0;
});

even;           // [10, 42]
even.length;    // 2
numbers.length; // 5
```

(See: `src/examples/js/filter.js`)

Mapping a Function Over an Array



Map Example

```
let strings = [
  "Mon, 14 Aug 2006 02:34:56 GMT",
  "Thu, 05 Jul 2018 22:09:06 GMT"
];

let dates = strings.map(function(s) {
  return new Date(s);
});

dates; // [Date, Date]
```

(See: `src/examples/js/map.js`)

Example: Folding an Array with `reduce`

```javascript
let a = [1, 2, 3];

// Sum numbers in `a'.
let sum = a.reduce(function(acc, elm) {
  // 1. `acc' is the accumulator
  // 2. `elm' is the current element
  // 3. You must return a new accumulator
  return acc + elm;
}, 0);

sum; // 6
```

(See: `src/examples/js/reduce.js`)

Exercise: Arrays and Functional Programming

1. Open the following file:
   src/www/js/array/array.js
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Hint: Use `https://developer.mozilla.org/` for documentation.

Partial Function Application and Currying

Introduction to Partial Function Application

- What happens when you call a function with fewer arguments than it was defined to take?
- Sometimes it's useful to provide fewer arguments and get back a function that accepts the remaining functions.

Simple Example Using Haskell

```haskell
-- Add two numbers:
add :: Int -> Int -> Int
add x y = x + y

-- Call a function three times:
tick :: (Int -> Int) -> [Int]
tick f = [f 1, f 2, f 3]

-- Prints "[11,12,13]"
main = print (tick (add 10))
```

Example Using the `bind` Method

```javascript
let add = function(x, y) {
  return x + y;
};

let add10 = add.bind(undefined, 10);

console.log(add10(2));
```

Exercise: Better Partial Functions

Write a `Function.prototype.curry` function that let's the following code work:

```
let obj = {
  magnitude: 10,

  add: function(x, y) {
    return (x + y) * this.magnitude;
  }.curry()
};

let add10 = obj.add(10);
add10(2); // Should return 120
```

- Use the following file: `src/www/js/partial/partial.js`

Scope and Context

Adding Context to a Scope

- We already discussed **scope**
  - Determines visibility of variables
  - Lexical scope (location in source code)
- There is also **context**
  - Refers to the location a function was invoked
  - Dynamic, defined at runtime
  - Context is accessible as the `this` variable

Calling Functions Through Objects

```
let apple  = {name: "Apple",  color: "red"   };
let orange = {name: "Orange", color: "orange"};

let logColor = function() {
  console.log(this.color);
};

apple.logColor  = logColor;
orange.logColor = logColor;

apple.logColor();
orange.logColor();
```

Context and the `this` Keyword

- The `this` keyword is a reference to “the object of invocation”
- Bound at invocation (depends on the call site)
- Allows a method to reference the “current” object
- A single function can then service multiple objects
- Central to prototypical inheritance in JavaScript

How JavaScript Sets the `this` Variable

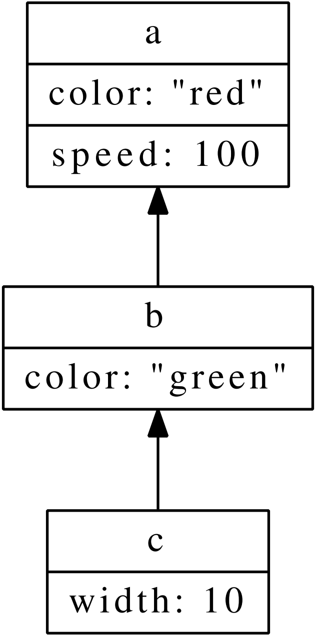
- Resides in the global binding
- Inner functions do not capture parent's `this` (there are several workarounds such as `let self = this;`, `bind`, and ES2015 arrow functions)
- The `this` object can be set manually! (Take a look at the `call`, `apply`, and `bind` functions.)

The Prototype

Inheritance in JavaScript

- JavaScript doesn't use classes, it uses prototypes
- There are ways to simulate classes (even ES2015 does it!)
- The prototypal model:
  - Tends to be smaller
  - Less redundant
  - Can simulate classical inheritance as needed
  - More powerful

Inheriting Properties from Other Objects

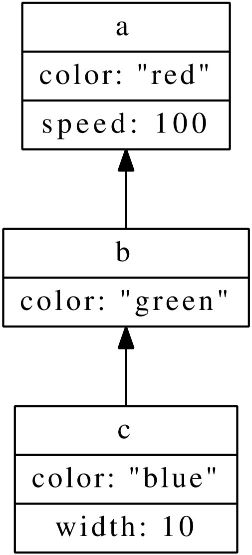



```
c.color === "green";
c.speed === 100;
```

Manual Configuration of Inheritance

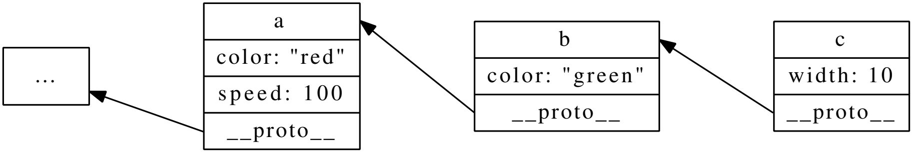
```
let a = {color: "red", speed: 100};
let b = Object.create(a);
let c = Object.create(b);

c.speed; // 100
```

Setting Properties and Inheritance




```
c.color = "blue";
c.color === "blue";
```

Inheritance with `__proto__`



Prototype Details

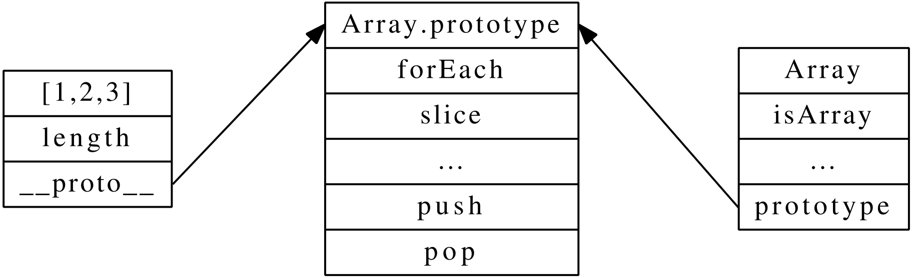
- All objects have an internal link to another object called its *prototype* (known internally as the `__proto__` property).
- The prototype object also has a prototype, and so on up the *prototype chain* (the final link in the chain is `null`).
- Objects *delegate* properties to other objects through the prototype chain.
- Only functions have a `prototype` property by default.

Using `__proto__` in ES2015

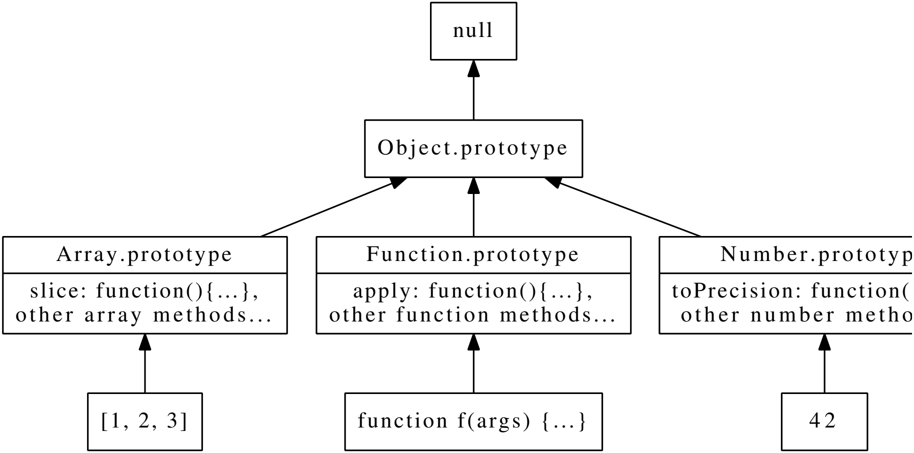
Starting in ECMAScript 2015, the `__proto__` property is standardized as an accessible property.

Warning: Using `__proto__` directly is strongly discouraged due to performance concerns.

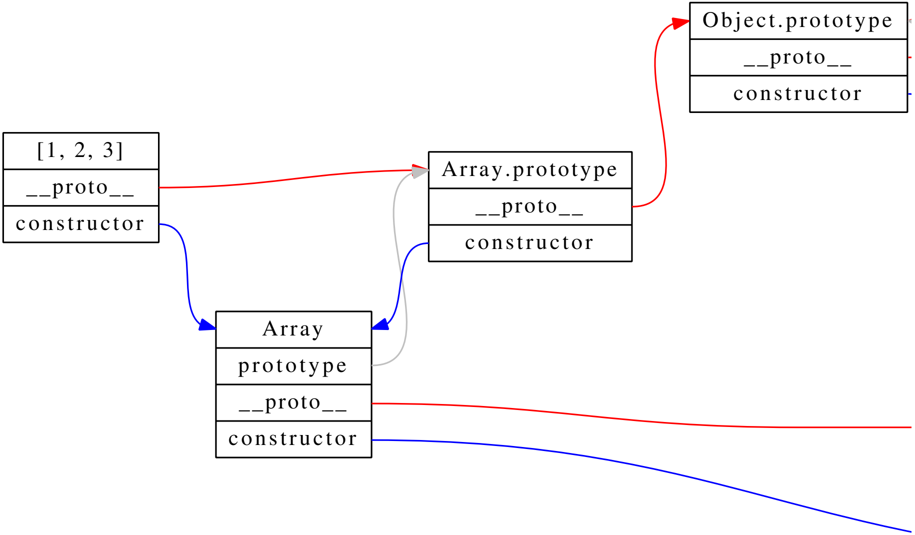
Looking at `Array` Instances



The Prototype Chain



Another Look at Array Instances



Establishing the Prototype Chain

Using `Object.create`

The `Object.create` function creates a new object and sets its `__proto__` property:

```
let a = {color: "red", speed: 100};
let b = Object.create(a);
let c = Object.create(b);
```

Using the `new` Operator

The `new` operator creates a new object and sets its `__proto__` property. The `new` operator takes a function as its right operand and sets the new object's `__proto__` to the function's `prototype` property.

```
let x = new Array(1, 2, 3);

// Is like:

let y = Object.create(Array.prototype);
y = Array.call(y, 1, 2, 3) || y;
```

Constructor Functions and Classes

Constructor Functions and OOP

```javascript
let Rectangle = function(width, height) {
  this.width  = width;
  this.height = height;
};

Rectangle.prototype.area = function() {
  return this.width * this.height;
};

let rect = new Rectangle(10, 20);
rect.area(); // 200
```

ES2015 Classes (Hidden Prototypes)

```
class Rectangle {
  constructor(width, height) {
    this.width  = width;
    this.height = height;
  }

  area() {
    return this.width * this.height;
  }
}

var rect = new Rectangle(10, 20);
rect.area(); // 200
```

Exercise: Constructor Functions

1. Open the following file:
   src/www/js/constructors/constructors.js
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Constructor Functions and Inheritance

```
let Square = function(width) {
  Rectangle.call(this, width, width);
};

Square.prototype = Object.create(Rectangle.prototype);
Square.prototype.sideSize = function() {return this.width;};

let sq = new Square(10);
sq.area(); // 100
```

ES2015 Classes and Inheritance

```
class Square extends Rectangle {
  constructor(width) {
    super(width, width);
  }

  sideSize() {
    return this.width;
  }
}

var sq = new Square(10);
sq.area(); // 100
```

Generic Functions (Static Class Methods)

Functions that are defined as properties of the constructor function are known as *generic* functions:

```
Rectangle.withWidth = function(width) {
  return new Rectangle(width, width);
};

let rect = Rectangle.withWidth(10);
rect.area(); // 100
```

ES2015 Static Class Methods

```
class Rectangle {
  constructor(width, height) {
    this.width  = width;
    this.height = height;
  }

  static withWidth(width) {
    return new Rectangle(width, width);
  }

  area() {
    return this.width * this.height;
  }
}

var rect = Rectangle.withWidth(10);
rect.area(); // 100
```

Property Descriptors

Setting property descriptors:

```
Object.defineProperty(obj, propName, definition);
```

- Define (or update) a property and its configuration
- Some things that can be configured:
  - `enumerable`: If the property is enumerated in `for .. in` loops
  - `value`: The property's initial value
  - `writable`: If the value can change
  - `get`: Function to call when value is accessed
  - `set`: Function to call when value is changed

Property Getters and Setters

```
function Car() {
  this._speed = 0;
}

Object.defineProperty(Car.prototype, "speed", {
  get: function() { return this._speed; },

  set: function(x) {
    if (x < 0 || x > 100) throw "I don't think so";
    this._speed = x;
  }
});

let toyota = new Car();
toyota.speed = 55; // Calls the `set' function.
```

ES2015 Getters and Setters

```
class Car {
  constructor() {
    this._speed = 0;
  }

  get speed() {
    return this._speed;
  }

  set speed(x) {
    if (x < 0 || x > 100) throw "I don't think so";
    this._speed = x;
  }
}

var toyota = new Car();
toyota.speed = 55; // Calls the `set speed' function.
```

Object-Oriented Programming: Gotcha

What's wrong with the following code?

```
function Parent(children) {
  this.children = [];

  // Add children that have valid names:
  children.forEach(function(name) {
    if (name.match(/\S/)) {
      this.children.push(name);
    }
  });
}

let p = new Parent(["Peter", "Paul", "Mary"]);
```

Accessing this via the bind Function

Notice where bind is used:

```
function ParentWithBind(children) {
  this.children = [];

  // Add children that have valid names:
  children.forEach(function(name) {
    if (name.match(/\S/)) {
      this.children.push(name);
    }
  }.bind(this));
}
```

Accessing this via a Closure Variable

Create an alias for `this`:

```
function ParentWithAlias(children) {
  let self = this;
  this.children = [];

  // Add children that have valid names:
  children.forEach(function(name) {
    if (name.match(/\S/)) {
      self.children.push(name);
    }
  });
}
```

Accessing `this` Directly via ES2015 Arrow Functions

Using the ES2015 *arrow function* syntax:

```
function ParentWithArrow(children) {
  this.children = [];

  // Add children that have valid names:
  children.forEach(name => {
    if (name.match(/\S/)) {
      this.children.push(name);
    }
  });
}
```

Introspection and Reflection

Simple Introspection Techniques

- The `instanceof` Operator:

```
// Returns `true':
[1, 2, 3] instanceof Array;
```

- The `Object.getPrototypeOf` Function:

```
// Returns `Array.prototype':
Object.getPrototypeOf([1, 2, 3]);
```

Object Mutability

Passing Objects to Functions

JavaScript uses *call by sharing* when you pass arguments to a function:

```
const x = {color: "purple", shape: "round"};

function mutator(someObject) {
  delete someObject.shape;
}

mutator(x);
console.log(x);
```

Produces:

```
{ color: 'purple' }
```

Object.freeze

```
Object.freeze(obj);

assert(Object.isFrozen(obj) === true);
```

- Can't add new properties
- Can't change values of existing properties
- Can't delete properties
- Can't change property descriptors

Object.seal

```
Object.seal(obj);

assert(Object.isSealed(obj) === true);
```

- Properties can't be deleted, added, or configured
- Property values can still be changed

Object.preventExtensions

```
Object.preventExtensions(obj);
```

- Prevent any new properties from being added

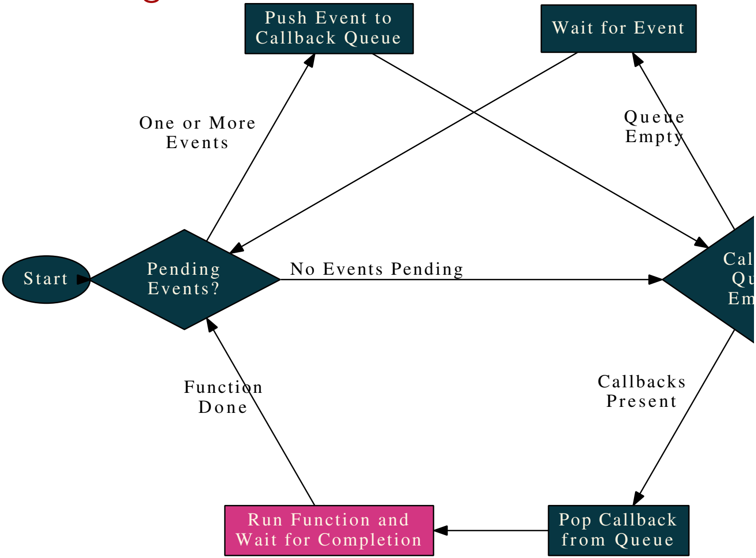
The JavaScript Runtime

Introduction to the Runtime

- JavaScript has a single-threaded runtime
- Work is therefore split up into small chucks (functions)
- Callbacks are used to divide work and call the next chunk
- The runtime maintains a work queue where callbacks are kept

(See the demo: `src/www/js/runtime/index.html`)

Visualizing the Runtime



(See the demo: `src/www/js/runtime/index.html`)

Loupe demo

Promises

Callbacks without Promises

```
$.getJSON("/a", function(data_a) {
  $.getJSON("/b/" + data_a.id, function(data_b) {
    $.getJSON("/c/" + data_b.id, function(data_c) {
      console.log("Got C: ", data_c);
    }, function() {
      console.error("Call failed");
    });
  }, function() {
    console.error("Call failed");
  });
}, function() {
  console.error("Call failed");
});
```

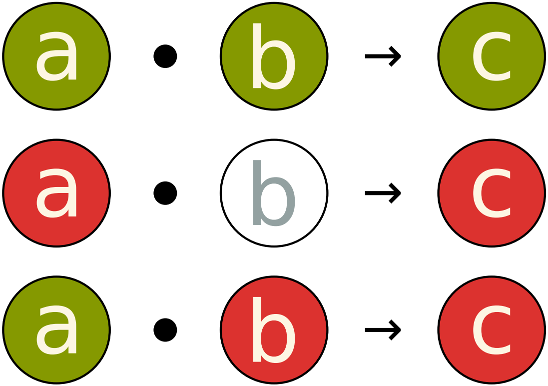
Callbacks Using Promises

```
$.getJSON("/a")
  .then(function(data) {
    return $.getJSON("/b/" + data.id);
  })
  .then(function(data) {
    return $.getJSON("/c/" + data.id);
  })
  .then(function(data) {
    console.log("Got C: ", data);
  })
  .catch(function(message) {
    console.error("Something failed:", message);
  });
```

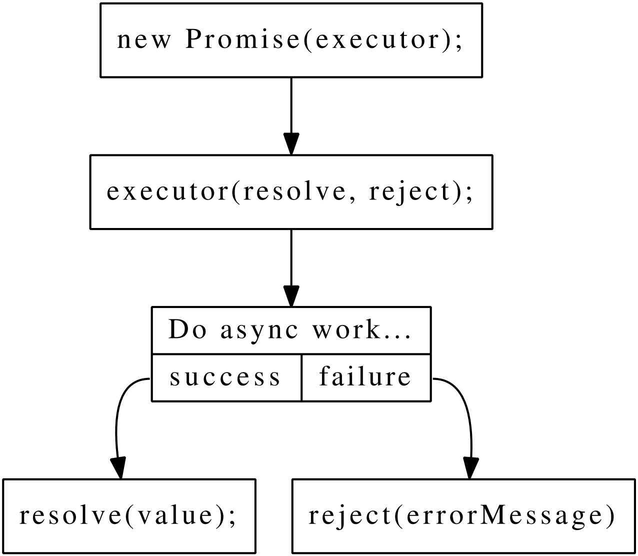
Promise Details

- Guarantee that callbacks are invoked (no race conditions)
- Composable (can be chained together)
- Flatten code that would otherwise be deeply nested

Visualizing Promises (Composition)



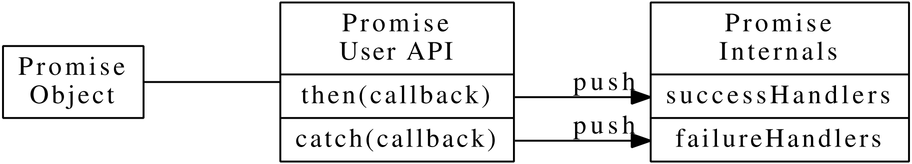
Visualizing Promises (Owner)



Example: Promise Owner

```
var delayed = function() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {

      if (/* some condition */ true) {
        resolve(/* resolved value */ 100);
      } else {
        reject(/* rejection value */ 0);
      }

    }, 500);
  });
};
```

Visualizing Promises (User)



Promise Composition Example

```
// Taken from the `src/spec/promise.spec.js' file.
var p = new Promise(function(resolve, reject) {
  resolve(1);
});

p.then(function(val) {
  expect(val).toEqual(1);
  return 2;
}).then(function(val) {
  expect(val).toEqual(2);
  done();
});
```

The Fetch API

Traditional XHR (Ajax) Requests

```
let req = new XMLHttpRequest();

req.addEventListener("load", function() {
  if (req.status >= 200 && req.status < 300) {
    console.log(req.responseText);
  }
});

req.addEventListener("error", function() {
  console.error("WTF?");
});

req.open("GET", "/example/foo.json");
req.send(/* data to send for POST, PATCH, etc. */);
```

Using the `fetch` Function

```
fetch("/api/artists", {credentials: "same-origin"})
  .then(function(response) {
    return response.json();
  })
  .then(function(data) {
    updateUI(data);
  })
  .catch(function(error) {
    console.log("Ug, fetch failed", error);
  });
```

Options and Results for `fetch`

```
fetch(url, {
  method: "POST",
  credentials: "same-origin",
  headers: {"Content-Type": "application/json; charset=utf-8"}
  body: JSON.stringify(data),
})
.then(function(response) {
  if (response.ok) return response.json();
  throw `expected ~ 200 but got ${response.status}`;
})
.then(console.log);
```

Browser Support

Browsers:

- IE (no support)
- Edge $>=$ 14
- Firefox $>=$ 34
- Safari $>=$ 10.1
- Chrome $>=$ 42
- Opera $>=$ 29

Using REST+JSON

- Fetch all artists (no body):

  `GET /api/artists`

- Fetch a single artist (no body):

  `GET /api/artists/2`

- Create a new artist (JSON body):

  `POST /api/artists`

- Update an artist (JSON body):

  `PATCH /api/artists/2`

- Delete an artist (no body):

  `DELETE /api/artists/2`

Exercise: Using the Fetch API

1. Start your server if it isn't running
2. Open `src/www/js/fetch/fetch.js`
3. Fill in the missing pieces
4. To test and debug, open

   `http://localhost:3000/js/fetch/`

The `async` and `await` Keywords

What are async Functions?

Functions marked as async become asynchronous and automatically return promises:

```
async function example() {
  return "Hello World";
}

example().then(function(str) {
  console.log(str); // "Hello World"
});
```

The `await` Keyword

Functions marked as `async` get to use the `await` keyword:

```
async function example2() {
  let str = await example();
  console.log(str); // "Hello World"
}
```

Question: What does the `example2` function return?

Example of async/await

```
async function getArtist() {
  try {
    let response1 = await fetch("/api/artists/1");
    let artist = await response1.json();

    let response2 = await fetch("/api/artists/1/albums");
    artist.albums = await response2.json();

    return artist;
  } catch(e) {
    // Rejected promises throw exceptions
    // when using `await'.
  }
}
```

An Even Better Example of async/await

```
async function getArtistP() {
  // Kick off two requests in parallel:
  let p1 = fetch("/api/artists/1").then(r => r.json());
  let p2 = fetch("/api/artists/1/albums").then(r => r.json());

  // Wait for both requests to finish:
  let [artist, albums] = await Promise.all([p1, p2]);

  artist.albums = albums;
  return artist;
}
```

Exercise: Using `async` and `await`

1. Start your server if it isn't running
2. Open `src/www/js/ajax/ajax.js`
3. Fill in the missing pieces
4. To test and debug, open

   `http://localhost:3000/js/ajax/`

WebSockets

WebSockets Basics

- Full duplex connection to a server
- Create your own protocol on top of WebSockets frames
- Not subject to the same origin policy (SOP) or CORS

How It Works

1. The browser requests that a new HTTP connection be *upgraded* to a raw TCP/IP connection
2. The server responds with `HTTP/1.1 101 Switching Protocols`
3. A simple binary protocol is used to support bi-directional communications between the client and server over the upgraded port 80 connection

Example: WebSockets

```javascript
let ws = new WebSocket("ws://localhost:3000/");

ws.onopen = function() {
  log("connected to WebSocket server");
};

ws.onmessage = function(e) {
  log("incoming message: " + e.data);
};

ws.send("PING");
```

(See: `src/www/js/apis/websockets/main.js`)

Security Considerations

- There are no host restrictions on WebSockets connections
- Encrypt traffic and confirm identity when using WebSockets
- Never allow foreign JavaScript to execute in a user's browser

Browser Support

- IE >= 10
- Firefox >= 6
- Safari >= 6
- Chrome >= 14
- Opera >= 12.10

Exercise: A Live Chatroom

1. Start your server if it isn't running
2. Open the following files:
   - `src/www/js/discography/components/chat.js`
   - `src/www/js/discography/index.html`
3. Fill in the missing pieces
4. Play with your chat room:

   `http://localhost:3000/js/discography/`

Web Storage

What is Web Storage?

- Allows you to store key/value pairs
- Two levels of persistence and sharing
- Very simple interface
- Keys and values *must* be strings

Session Storage

- Lifetime: same as the containing window/tab
- Sharing: Only code in the same window/tab
- 5MB user-changeable limit (10MB in IE)
- Basic API:

```
sessionStorage.setItem("key", "value");
let item = sessionStorage.getItem("key");
sessionStorage.removeItem("key");
```

Local Storage

- Lifetime: unlimited
- Sharing: All code from the same domain
- 5MB user-changeable limit (10MB in IE)
- Basic API:

```
localStorage.setItem("key", "value");
let item = localStorage.getItem("key");
localStorage.removeItem("key");
```

The Storage Object

Properties and methods:

- `length`: The number of items in the store.
- `key(n)`: Returns the name of the key in slot `n`.
- `clear()`: Remove all items in the storage object.
- `getItem(key)`, `setItem(key, value)`, `removeItem(key)`.

Browser Support

- IE >= 8
- Firefox >= 2
- Safari >= 4
- Chrome >= 4
- Opera >= 10.50

Exercise: Chatroom Replay

1. Start your server if it isn't running
2. When receiving an incoming message from the chat server cache the message in the `sessionStorage`.
3. When the page first loads insert all of the cached chat messages into the UI.
4. Open the following files:
   - `src/www/js/discography/components/chat.js`
5. Fill in the missing pieces
6. Send some chat messages then reload:

   `http://localhost:3000/js/discography/`

Service Workers

Service Worker Basics

- Intended to replace AppCache
- Can intercept network requests and decide how to respond (make real request, pull from cache, etc.)
- Can cache all assets when started
- Allows for complete offline experience

Registering a Service Worker

From your site's JavaScript:

```
navigator.serviceWorker.register("worker.js")
  .then(function(registration) {
    console.log("registration complete");
  })
  .catch(function(error) {
    console.log("ERROR: " + error);
  });
```

(See `src/www/js/apis/serviceworkers/main.js`)

Caching Resources

```
self.addEventListener("install", function(event) {
  console.log("installed");

  async function ready() {
    let cache = await caches.open('myCacheName');
    await cache.addAll(["/api/artists"]);
    self.skipWaiting(); // activate a new version.
  }

  event.waitUntil(ready());
});
```

(See `src/www/js/apis/serviceworkers/worker.js`)

Additional Uses of Service Workers

- Push notifications for mobile and desktop
- Background sync (wait for network connection, then send a request)
- Installable Web Apps (web apps that act like native mobile applications)
- Work with a Transactional High-Performance Key-Value Store

Browser Support

- IE (no support)
- Edge >= 17 (2015)
- Firefox >= 44.0 (2016)
- Safari >= 11.1 (2018)
- Chrome >= 40 (2015)
- Opera >= 27 (2015)

Web Workers

Web Worker Basics

- Allows you to start a new background "thread"
- Messages can be sent to and from the worker
- Message handling is done through events
- Load scripts with: `importScripts("name.js");`

Browser Support

- IE $>=$ 10
- Firefox $>=$ 3.5
- Safari $>=$ 4
- Chrome $>=$ 4
- Opera $>=$ 10.6

Node.js

Node.js

- Server-side JavaScript engine
- Also provides a general-purpose environment
- Write servers, or GUI programs in JavaScript
- Most development tools are written in JavaScript and use Node.
- `https://nodejs.org/`

Node Package Manager (npm)

- Repository of JavaScript libraries, frameworks, and tools
- Tool to create or install packages
- Run scripts or build processes
- 800k+ packages available
- If it has something to do with JavaScript you install it with npm
- `https://www.npmjs.com/`

Introduction to TypeScript

What is TypeScript

- A language based on ESNEXT
- Compiles to ES5
- Contains the following additional features:
  - Types and type inference!
  - Generics (polymorphic types)
  - Interfaces and namespaces
  - Enums and union types

Type Annotations

```typescript
function add(x: number, y: number): number {
  return x + y;
}
```

Type Checking

```
// Works!
const sum = add(1, 2);

// error: Argument of type '"1"' is not assignable
// to parameter of type 'number'.
add("1", "2");
```

Type Inference

```
// Works!
const sum = add(1, 2);

// error: Property 'length' does not exist
// on type 'number'.
console.log(sum.length);
```

Additional Examples

Look in the following folder for additional examples:

`src/www/js/alternatives/typescript/examples`

Linting Tools

Introduction to Linting Tools

- Linting tools parse your source code and look for problems
- The two most popular linters for JavaScript are JSLint and ESLint
- ESLint is about 3x more popular than JSLint

About ESLint

- Integrates with most text editors via plugins
- Fully configurable, easy to add custom rules
- Enforce project style guidelines

Using ESLint Manually

```
$ npm install -g eslint
$ eslint yourfile.js
```

ESLint Plugins

- Visual Studio Code
- Sublime Text
- Emacs
- vim
- Official Integration List

Transpiling with Babel

Introduction to Babel

- Automated JavaScript restructuring, refactoring, and rewriting
- Parses JavaScript into an Abstract Syntax Tree (AST)
- The AST can be manipulated in JavaScript
- Includes *presets* to convert from one form of JavaScript to another
    - ESNEXT to ES5
    - React's JSX files to ES5
    - Vue's VUE files to ES5
    - etc.

Manually Using Babel

Process all files from the `input` directory and put all generated files in the `output` directory:

```
$ npm install --save-dev babel-cli babel-preset-env
$ ./node_modules/.bin/babel --presets env -d output input
```

(Note: Babel 7 will use a slightly different command line.)

Integrating Babel with Your Build Tools

Most build tools (Grunt, Gulp, Webpack) support a Babel phase.

Simple overview of a build process:

1. Gather up all necessary JavaScript files
2. Run the files through a linter like ESLint
3. Concatenate them into a single file in the right order
4. Run that file through Babel
5. Minify and compress the file Babel produced

Packaging with Webpack

What is Webpack?

Webpack is a build tool for web applications:

- Uses ES2015 modules to bundle JavaScript into a single file ready for deployment to production
- Transpiles JavaScript (i.e. ES20* to ES5)
- Lint code and run tests
- Bundles many types of assets (CSS, HTML templates, etc.)
- Can load remote assets on-demand

Exporting and Importing Identifiers

- Export identifiers from a library:

```
const magicNumber = 42;

function sayMagicNumber() {
  console.log(magicNumber);
}

export { sayMagicNumber };
```

- Import those identifiers elsewhere:

```
import sayMagicNumber from './module.js';
sayMagicNumber();
```

Explicit Dependencies in JavaScript

When using ES2015 modules:

- Dependencies are explicit through imports
- Removes global namespace pollution
- You can import part of a library, or the entire thing
- Strict mode enabled by default

Bundling JavaScript Modules

Webpack will:

1. Start with your main JavaScript file
2. Follow all `import` statements
3. Generate a single file containing all JavaScript

The generated file is know as a *bundle*.

More Power Through Loaders

Webpack becomes a full build tool via *loaders*. Here are some example loaders:

`babel-loader` Transpiles JavaScript using Babel
`eslint-loader` Lints JavaScript using ESLint
`mocha-loader` Run tests before building
`html-loader` Bundle HTML templates
`sass-loader` Process and bundle Sass

Configuring Webpack

Webpack is configured through a JavaScript file named `webpack.config.js`. Using this file you can:

- Tell Webpack what file is the main JavaScript file
- Specify which loaders you are using and in which order
- Add additional JavaScript snippets such as polyfills to the bundle
- Go crazy since you are writing in JavaScript

Webpack Demonstration

Let's take a look at a Webpack demonstration application:

1. Open the following folder in your text editor:
   src/www/js/tools/webpack
2. Review the example files:
   - `index.html`
   - `src/index.js`
   - `src/template.html`
   - `webpack.config.js`
3. Build the application with:
   $ npm run build

If you are running your Node.js server you can access this application at `http://localhost:3000/js/tools/webpack/`

Testing Overview

3 Types of Tests

1. Unit
2. Integration
3. End-to-End (E2E)

Unit and **Integration** tests can be run without a browser. Faster to run, sometimes slower to write.

E2E tests simulate user behavior interacting with a browser environment. Slower to run, sometimes faster to write.

Unit and Integration Tests

Most popular framework is **Jest**.

Other common frameworks are **Mocha**, **Jasmine**, **AVA**, **Tape**, and **QUnit**

Unit and Integration Tests Categories

There's two basic categories that JS unit tests fall into:

1. Pure JavaScript
2. JavaScript + Browser

Code that is "just" JavaScript (no browser APIs) is the easiest to test.

Testing code that includes the browser is often challenging and often requires more mocking.

Jest: Basics

What is Jest?

- JS testing framework
- Focus on simplicity and easy configuration
- Easy mocking of modules
- Good for unit and integration tests

Example: Writing Jest Tests

```javascript
const add = (x, y) => x + y

describe('#add', () => {
  it('adds two numbers together', () => {
    expect(add(1, 2)).toEqual(3)
  })
})
```

Running Jest Tests

1. `yarn add jest`
2. Make a `*.spec.js` file
3. Run `yarn jest`

Yes, it's just that easy.

Jest: Expect & Matchers

Most Common Matchers

`toEqual(val)`: Most common equality matcher. Compares objects or arrays by comparing contents, not identity.

`toMatch(/hello/)`: Tests against regular expressions or strings.

Expecting an Error

`toThrow(message)`: Tests the function will throw an error.

```
describe('#findById', () => {
  it('should throw if not a number', () => {
    expect(() => findById('invalid'))
      .toThrow('Must provide a number')
  })
})
```

Expecting the Opposite

You can chain `not` to test the opposite

```
it('test the opposite', () => {
  expect(0).not.toEqual(1)
})
```

Other Matchers Sometimes Used

`toContainEqual(x)`: Expect an array to contain `x`.

`toBe(x)`: Compares with `x` using ===.

`toBeTruthy()`: Should be true `true` when cast to a Boolean.

`toBeFalsy()`: Should be `false` when cast to a Boolean.

`arrayContaining(array)`: Checks it's a subset (order doesn't matter).

Exercise: Writing a Test with Jest

1. Open `src/www/js/jest/__tests__/adder.spec.js`
2. Do exercise 1
3. To test and debug, run

```
cd src
yarn test www/js/jest/__tests__/adder.spec.js
```

Jest: Environment & Life Cycle

Jest Environment

- Each spec file runs in its **own, isolated environment**
- `setupTestFrameworkScriptFile`: Shared, one-time setup

```json
{
  "jest": {
    "setupTestFrameworkScriptFile": "<rootDir>/setup_jest.js"
  }
}
```

- `setupFiles`: Setup files run once before every test file

Life Cycle Callbacks

Each of the following functions takes a callback as an argument:

`beforeEach`: Before each `it` is executed.
`beforeAll`: Once before any `it` is executed.
`afterEach`: After each `it` is executed.
`afterAll`: After all `it` specs are executed.

Abstracting Life Cycle Callbacks

These functions can be invoked from any module, as long as the calling context is within a spec file!

```javascript
// setup.js

const startWithLoggedInUser = () => {
  beforeEach(() => {
    // set up your app state to simulate a logged-in user
  })

  afterEach(() => {
    // clean up app state...
  })
}
```

Abstracting Life Cycle Callbacks Use

```js
// todos.js

describe('user todos', () => {
  startWithLoggedInUser()

  it('should read user todos', () => { /* ... */ })
})
```

Jest: Pending Tests

Pending Tests

Tests can be marked as pending:

```
it.todo('should do a thing')
```

Jest: Spies

What Are Spies

- Spies allow you to track calls to a method
  - Arguments
  - Results
- Passes call through to original implementation

Spies API

Creating a spy:

```
const spy = jest.spyOn(myObject, 'method')
```

Removing a spy:

```
spy.restore()
```

Spying on a Function or Callback (Call Tracking)

```
const video = {
  play() { return true },
}

it('should play a video', () => {
  const spy = jest.spyOn(video, 'play')
  const isPlaying = video.play()

  expect(spy).toHaveBeenCalled()
  expect(isPlaying).toBe(true)

  spy.mockRestore()
})
```

Spying on a Function or Callback (Call Fake)

```
it('should allow a fake implementation', () => {
  const spy = jest.spyOn(video, 'play')
    .mockImplementation(() => false)
  const isPlaying = video.play()

  expect(spy).toHaveBeenCalled()
  expect(isPlaying).toBe(false)

  spy.mockRestore()
})
```

Exercise: Using Jest Spies

1. Open `src/www/js/jest/__tests__/adder.spec.js`
2. Read the code then do exercise 2
3. To test and debug, run

```
cd src
yarn test www/js/jest/__tests__/adder.spec.js
```

Jest: Mocks

Mocks

- Mocks are functions with pre-programmed behavior
- Can be used to replace methods or module dependencies
- Why mock
  - Avoid expensive / slow calls (server calls, complex compuations, etc.)
  - Simplifies dependencies
  - Avoid complex test setup (e.g. dependency requires a bunch of state)
  - You follow the “London-TDD” style

Mocks API

Creating a mock:

```
const mock = jest.fn()
```

With behavior:

```
const mock = jest.fn(() => 'yay')

mock() // 'yay'
```

Mock Functions

Say we're testing a higher-order function:

```
const forEach = (items, callback) => {
  for (let i = 0; i < items.length; i++) {
    callback(items[i])
  }
}
```

Captures Calls

You can create a mock function to capture calls.

```
const myMock = jest.fn()
```

Example:

```
it('capture calls', () => {
  const mockCallback = jest.fn()
  forEach([0, 1], mockCallback)

  expect(mockCallback.mock.calls.length).toEqual(2)
  expect(mockCallback.mock.calls).toEqual([[0], [1]])
})
```

Captures all arguments

```
const myMock = jest.fn()
myMock('hello', 'world')
myMock(1, 2, 3)
expect(myMock.mock.calls).toEqual([
  ['hello', 'world'],
  [1, 2, 3],
])
```

Provide Fake Behavior

You can specify static behavior of a mock function.

```
const getUserName = (id, lookupUser) => {
  const user = lookupUser(id)
  return user.name
}

it('should specify behavior', () => {
  const mockFn = jest.fn(() => ({
    id: 1,
    name: 'Andrew'
  }))
  expect(getUserName(1, mockFn))
    .toEqual('Andrew')
})
```

Provide Dynamic Behavior

You can use the arguments to a mock function to create dynamic behavior.

```
const getUserNames = (ids, lookupUser) => (
  map(compose(prop('name'), lookupUser), ids)
  // aka: ids.map(lookupUser).map(user => user.name)
)

it('should handle dynamic behavior', () => {
  const mockUsers = {
    1: { id: 1, name: 'Andrew' },
    2: { id: 2, name: 'Billy' },
    3: { id: 3, name: 'Charlie' },
  }
  const mockLookup = jest.fn((id) => mockUsers[id])

  expect(getUserNames([1, 3], mockLookup))
    .toEqual(['Andrew', 'Charlie'])
})
```

Mock Return Values

```
it('should mock return values', () => {
  const mock = jest.fn()
    .mockReturnValueOnce(42)
    .mockReturnValueOnce('hello')
    .mockReturnValue('default')

  expect(mock()).toEqual(42)
  expect(mock()).toEqual('hello')
  expect(mock()).toEqual('default')
})
```

Cleanup per mock

- **mockClear**: reset calls/results
- **mockReset**: `mockClear` + reset return values / implementations
- **mockRestore**: `mockReset` + restores original non-mocked implementation (for spies)

Cleanup in beforeEach

- **jest.clearAllMocks**
- **jest.resetAllMocks**
- **jest.restoreAllMocks**

Cleanup in config

Can provide `package.json` config to do it for **all** tests:

```json
{
  "jest": {
    "clearMocks": true,
    "resetMocks": true,
    "restoreMocks": true
  }
}
```

Jest: Timers

Testing Time-Based Logic (The Setup)

Given a delay function:

```
const delay = (ms, fn) => {
  setTimeout(fn, ms)
}
```

This won't work the way you want:

```
it('will not wait for no one', () => {
  const mock = jest.fn()
  delay(1000, mock)
  expect(mock).toHaveBeenCalled() // FAILS
})
```

Why?

The Trouble With Time

JavaScript is a single-threaded runtime environment.

Tests run synchronously.

Mocking Time

Set up with

```
jest.useFakeTimers()
```

Many ways to manipulate time:

`jest.runAllTimers()`: Run all timers until there are none left
`jest.runOnlyPendingTimers()`: Run currently pending timers
`jest.advanceTimersByTime(ms)`: Advance all timers by `ms`
`jest.clearAllTimers()`: Clear all pending timers

Running All Timers

```javascript
jest.useFakeTimers()

it('should all timers', () => {
  const mock = jest.fn()
  delay(1000, mock)
  jest.runAllTimers()
  expect(mock).toHaveBeenCalled()
})
```

Running Pending Timers

Given

```
const delayInterval = (ms, fn) => {
  setInterval(fn, ms)
}
```

Using `jest.runAllTimers()` will run forever.

Use `jest.runOnlyPendingTimers()` instead.

Running Pending Timers (Example)

```
it('should run pending timers', () => {
  const mock = jest.fn()
  delayInterval(1000, mock)
  jest.runOnlyPendingTimers()
  expect(mock).toHaveBeenCalled()
})
```

Advancing By Time

```
it('should advance time', () => {
  const mock = jest.fn()
  delay(1000, mock)
  jest.advanceTimersByTime(999)
  expect(mock).not.toHaveBeenCalled()
  jest.advanceTimersByTime(1)
  expect(mock).toHaveBeenCalled()
})
```

Cleanup

Good idea to use

```
afterEach(() => {
  jest.clearAllTimers()
})
```

Safer Setup

`jest.useFakeTimers` impacts all tests in a test file.

Using fake timers can have unforeseen consequences:

- Promises behave unexpectedly
- `async`/`await` behaves unexpectedly

Instead, you can tell each test to use *real* timers and create a way to set up a fake timer.

Safer Setup (Setup)

```
export const setupForFakeTimers = () => {
  beforeEach(() => {
    jest.useRealTimers()
  })

  return () => jest.useFakeTimers()
}
```

Safer Setup (Example)

```
describe('sometimes faking timers', () => {
  const useFakeTimers = setupForFakeTimers()

  it('normally has real timers', () => {
    // jest.runAllTimers() <-- does not work
  })

  it('should have a fake timer', () => {
    useFakeTimers()
    jest.runOnlyPendingTimers()
  })
})
```

Jest: Async

Testing Asynchronous Functions

Given a (fake) server interaction:

```
const users = {
  1: { id: 1, name: 'Andrew' },
  2: { id: 2, name: 'Billy' },
}

const getUser = (id) => new Promise((res, rej) => {
  process.nextTick(() => (
    users[id]
      ? res(users[id])
      : rej('User ID ' + id + ' not found.')
  ))
})
```

Testing Asynchronous Functions (with async)

You can use an `async` callback for `it`:

```
it('should handle async', async () => {
  const user = await getUser(1)
  expect(user).toEqual({ id: 1, name: 'Andrew' })
})
```

Or more tersely with `await expect(...).resolves`:

```
it('should handle async', async () => {
  return await expect(getUser(1))
    .resolves.toEqual({ id: 1, name: 'Andrew' })
})
```

Testing Asynchronous Functions (with Promises)

If `async` isn't available, you could return a promise:

```
it('should handle async', () => {
  return getUser(1)
    .then((res) => {
      expect(res).toEqual({ id: 1, name: 'Andrew' })
    })
})
```

You can make it more terse with `expect(...).resolves`:

```
it('should handle async', () => {
  return expect(getUser(1))
    .resolves.toEqual({ id: 1, name: 'Andrew' })
})
```

Testing Async Dependencies

Say we're testing a function that uses our async `getUser` function indirectly:

```
const getUserName = async (id) => {
  const user = await getUser(id)
  return user.name
}

it('can still await with resolves', async () => {
  return await expect(getUserName(2))
    .resolves.toEqual('Billy')
})
```

Why does this work?

Testing Inaccessible Async Operations

Sometimes we do something async but don't await its result:

```
it('is hard to find how to wait!', async () => {
  const mockFn = jest.fn()
  await loadUserInBackground(1, mockFn) // won't wait!
  expect(mockFn)
    .toHaveBeenCalledWith({ id: 1, name: 'Andrew' })
})

// Test output FAILURE:
//
// Expected: {"id": 1, "name": "Andrew"}
// Number of calls: 0
```

Testing Inaccessible Async Operations

Easiest way is to force a process tick in the test.

We call it “flushing promises”.

```
const flushPromises = () => (
  new Promise(res => process.nextTick(res))
)
```

Testing Inaccessible Async Operations (Example)

```
it('can have promises flushed', async () => {
  const mockFn = jest.fn()
  loadUserInBackground(1, mockFn)
  await flushPromises()
  expect(mockFn)
    .toHaveBeenCalledWith({ id: 1, name: 'Andrew' })
})
```

This happens all the time in UI unit testing, e.g. with React.

Async Error Handling

When you reject a promise and don't catch it correctly…

```
it('should fail', () => {
  return getUser(42)
    .then((res) => { expect(1).toEqual(1) })
})
```

Your test will fail:

```
Error: Failed: "User ID 42 not found."
```

Async Error Handling (with async)

You can test for error handling with `async`/`await`:

```
it('should catch errors', async () => {
  try {
    await getUser(42)
  } catch (e) {
    expect(e).toEqual('User ID 42 not found.')
  }
})
```

Async Error Handling (Silent Failures)

Unfortunately, if the promise *doesn't* reject, the assertion is never called!

```
it('does not fail :-(', async () => {
  try {
    await getUser(1)
  } catch (e) {
    expect(1).toEqual(0) // Still passes!
  }
})
```

Async Error Handling (with rejects)

Safest approach is to use `expect(...).rejects`:

```
it('should return error message', async () => {
  await expect(getUser(42))
    .rejects.toEqual('User ID 42 not found.')
})
```

Async Error Handling (with rejects FTW)

This will correctly fail the test if the promise was not rejected:

```
it('should fail', async () => {
  await expect(getUser(1))
    .rejects.toEqual('User ID 42 not found.')
})

// Test output:
//
// Received promise resolved instead of rejected
// Resolved to value: {"id": 1, "name": "Andrew"}
```

Async Error Handling (thrown Errors)

If you `throw` an error, you must write a different expectation.

```
const boom = async () => {
  throw new Error('kaboom')
}

it('will not match :-(', async () => {
  return await expect(boom())
    .rejects.toEqual('kaboom')
})

// Test output FAILURE
// Expected: "kaboom"
// Received: [Error: kaboom]
```

Async Error Handling (with toThrow)

Use `toThrow` instead:

```
const boom = async () => {
  throw new Error('kaboom')
}

it('will match with toThrow', async () => {
  return await expect(boom())
    .rejects.toThrow('kaboom')
})
```

Quick Note About Fake Async…

`setTimeout(cb, 0)` and `process.nextTick(cb)` are **not the same thing**.

setTimeout "takes longer" than `process.nextTick`

```
const flushPromises = () => (
  new Promise(res => process.nextTick(res))
)

it('will not work', async () => {
  const mockFn = jest.fn()
  setTimeout(mockFn, 0)
  await flushPromises()
  expect(mockFn).toHaveBeenCalled() // Nope.
})
```

Prefer process.nextTick

When possible, mock async behavior with `process.nextTick`.

Turns out `jest.useFakeTimers()` messes with `setTimeout` behavior...

```
const flushPromisesSTO = () => (
  new Promise(res => setTimeout(res, 0))
)
```

setTimeout Gets Weird

```
it('does not work :-(', async () => {
  jest.useFakeTimers()
  const mockFn = jest.fn()
  setTimeout(mockFn, 0)
  await flushPromisesSTO()
  expect(mockFn).toHaveBeenCalled()
})

// Test output FAILURE:
// Timeout - Async callback was not invoked within
// the 5000ms timeout
```

No Problems with process.nextTick

```
it('does work', async () => {
  jest.useFakeTimers()
  const mockFn = jest.fn()
  process.nextTick(mockFn)
  await flushPromises()
  expect(mockFn).toHaveBeenCalled() // Yep!
})
```

Save yourself the pain and stick with `process.nextTick` when you can.

Exercise: Handling Async Functions

1. Open `src/www/js/jest/__tests__/async.spec.js`
2. Do the exercises
3. To test and debug, open

```
cd src
yarn test www/js/jest/__tests__/async.spec.js
```

Testing JS + Browser

Testing Browser Interactions in Unit Tests

Sometimes your unit/integration tests will involve browser APIs, e.g.:

- `addTodoToDOMList`: appends an `li` element to a `ul` todos element.

Use **jsdom**: creates fake browser environment

DOM Manipulation

```
const addTodoToDOMList = (text) => {
  const todos = document.getElementById('todos')

  const todo = document.createElement('li')
  todo.appendChild(document.createTextNode(text))

  todos.appendChild(todo)
}
```

Testing DOM Manipulations Setup

Set the browser body *each time*, it persists between tests.

```
beforeEach(() => {
  // set up the browser DOM
  document.body.innerHTML = '<ul id="todos"></ul>'
})
```

Testing DOM Manipulations

```
it('should add a todo to the todos', () => {
  addTodoToDOMList('Learn jsdom')
  addTodoToDOMList('Practice DOM changes')

  const todos = document.getElementById('todos')
  const todosText = Array.from(todos.children)
    .map(child => child.textContent)

  expect(todosText).toEqual([
    'Learn jsdom',
    'Practice DOM changes',
  ])
})
```

Pure magic.

UI Testing Libraries

UI Testing Libraries

Makes it easier to write UI tests.

- DOM-only
  - `@testing-library/dom`
- React
  - `@testing-library/react`
  - `enzyme`
  - `react-test-renderer`
- Vue
  - `@testing-library/vue`

Unit Testing Best Practices

Most Importantly

- Practice TDD
  1. Red (write a failing test)
  2. Green (make the test pass)
  3. Refactor (make your code better)

Really. Just do it.

Be Persistent and Track Your Discoveries

- There are also hard, tricky testing situations. Don't give up.
- Google, Stack Overflow, ask teammates, ping `@andrewsouthpaw`, etc.
- Track solutions in `test-helpers.js`
  - e.g.: `flushPromises`, `stubTime`
- Keep a living document of testing style and troubleshooting.

Other Valuable Practices

- Write abstractions to make your test files easier to read
- Make factories to simplify test data creation
  - e.g. `newTodo`, `newUser`, `newAppState`, etc.
- Test for error handling on server interactions
- Automate your tests so they run all the time

Mock Less, Smile More

- Avoid mocking/stubbing as they create implicit interface contracts. Generally only mock:
    1. Server calls
    2. Complex functions / behavior
    3. Slow / expensive functions
- Mocking reduces confidence in system actually working
- Mocking is often hard to read

UI Testing Best Practices

- Separate business logic from DOM manipulation
- Interact with UI as a user, not a programmer, e.g.:
  - Click the "Save" button, don't invoke `handleSaveClick`
  - Expect text changes, not `<p>` elements

E2E Testing

E2E Testing

It simulates a user interacting with your website via a browser.

- PROS: Less mocking –> easier to write
- CONS: Slow to run

E2E Testing Frameworks

Popular services/frameworks:

- Cypress
- Nightwatch
- Selenium

Compatibility Testing

Compatibility Testing

Depending on your team's requirements, you may need to make sure your site works in all browsers.

Popular services:

- SauceLabs
- BrowserStack
- LambdaTest

These tests are the most expensive to write and maintain.