Intermediate JavaScript

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

DEVALOT

Overview

What's In Store

Day 1	Day 2
Quick Review	Web Components
Functional Programming	WebSockets
Prototypes and Classes	WebStorage
The Asynchronous Runtime	Service Workers
Promises and `await`	Developer Tools
The `fetch` API	Testing w/ Jasmine

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?

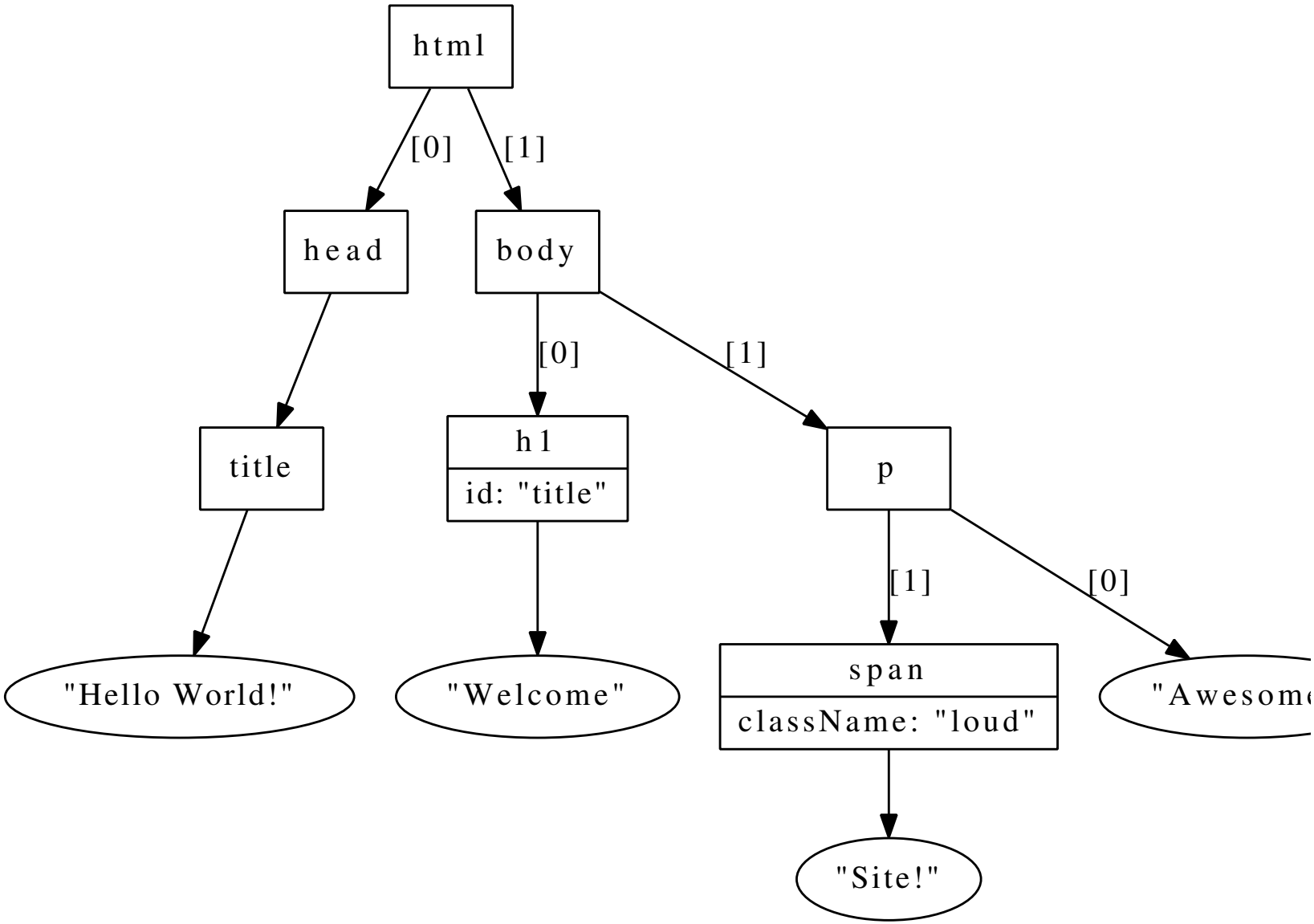
- What most people hate when they say they hate JavaScript
- The DOM is the browser's API for the document
- Through it you can manipulate the document
- Browser parses HTML and builds a tree structure
- It's a live data structure

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:

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

- Maps to:

```
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 eteration or web workers
- Eteration: 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)

```
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)

```
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

```
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:
  - via the TypeScript compiler
  - 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

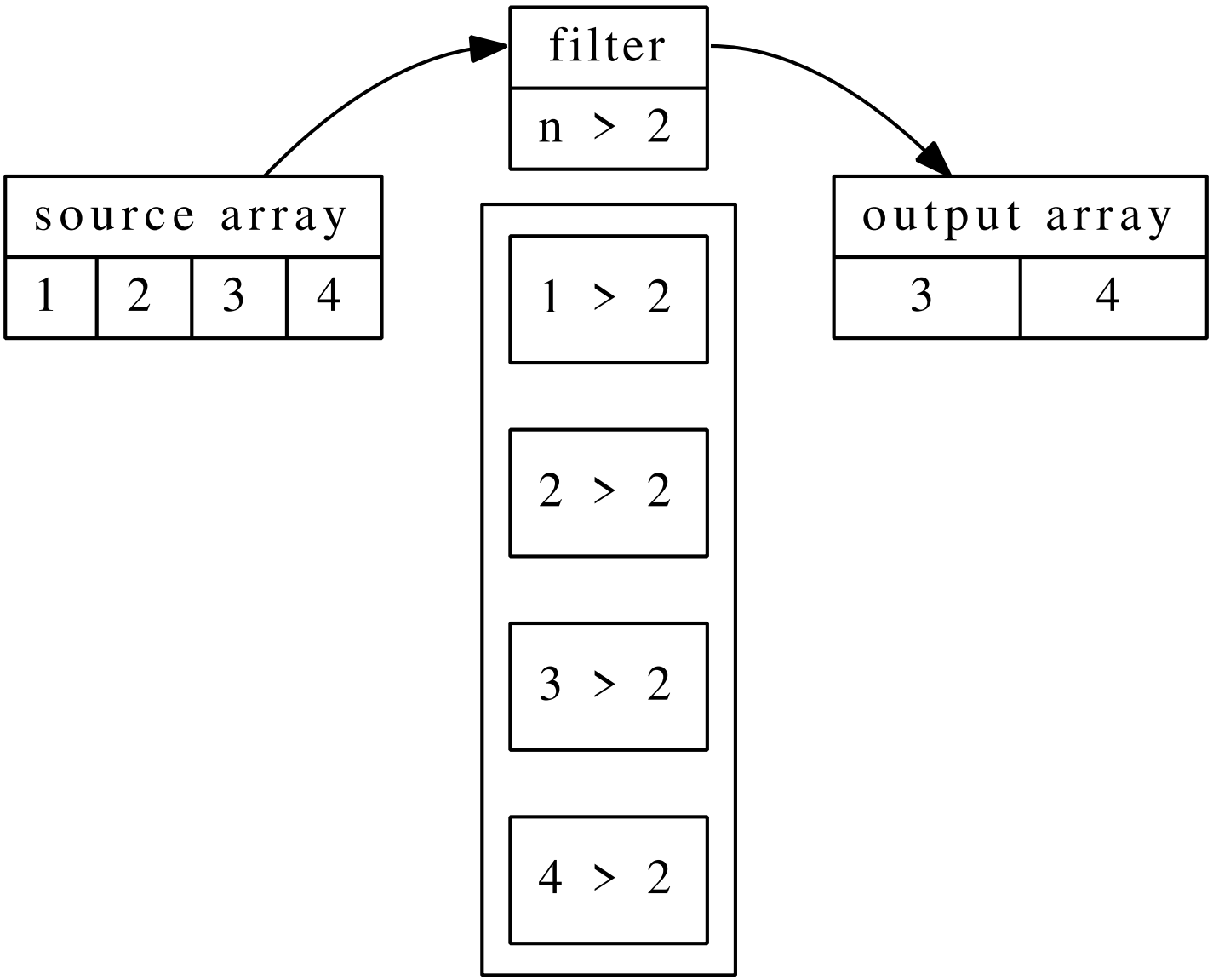
- Test if a function returns `true` on all elements:

```
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

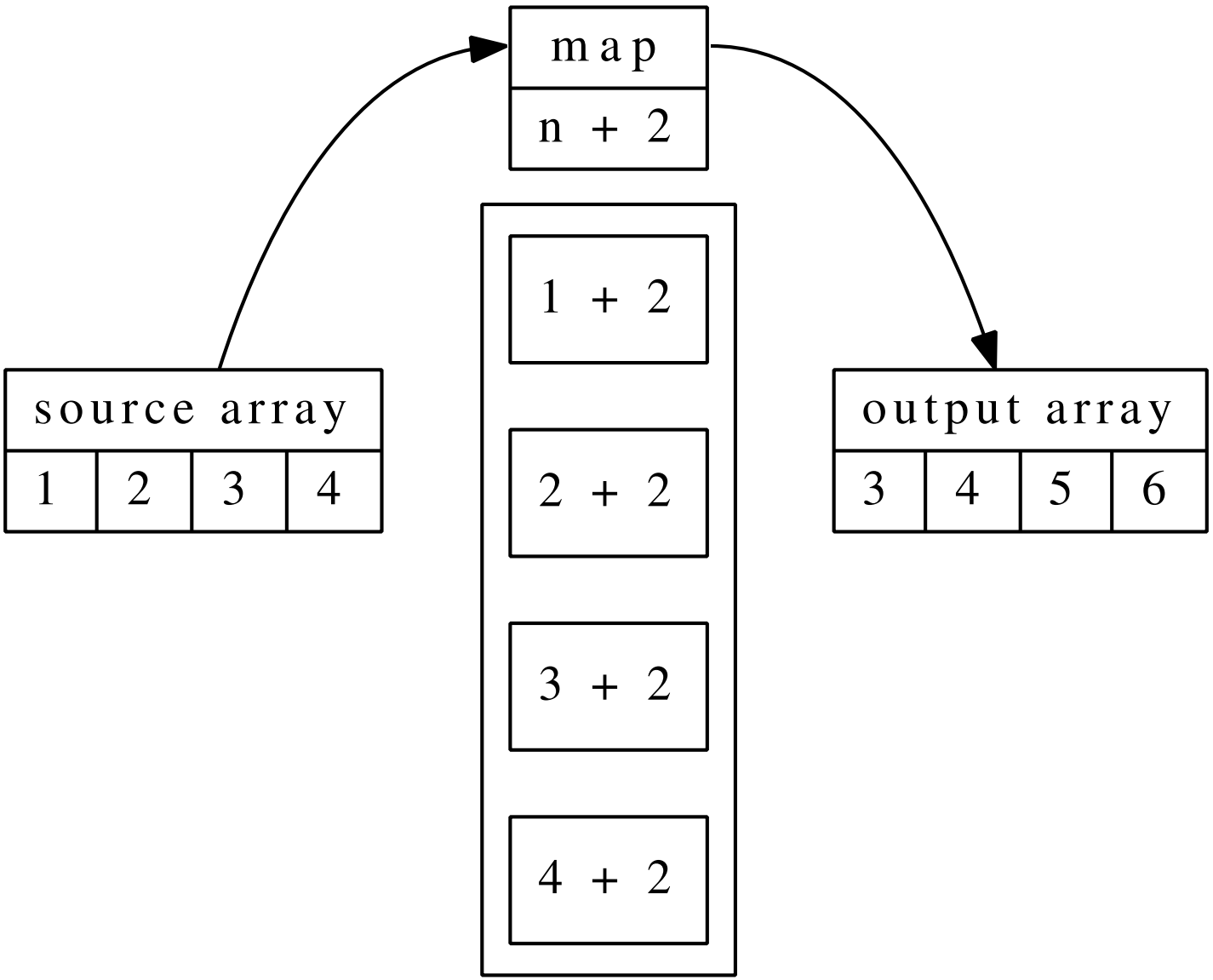


Filter Example

```
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`

```
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

```
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

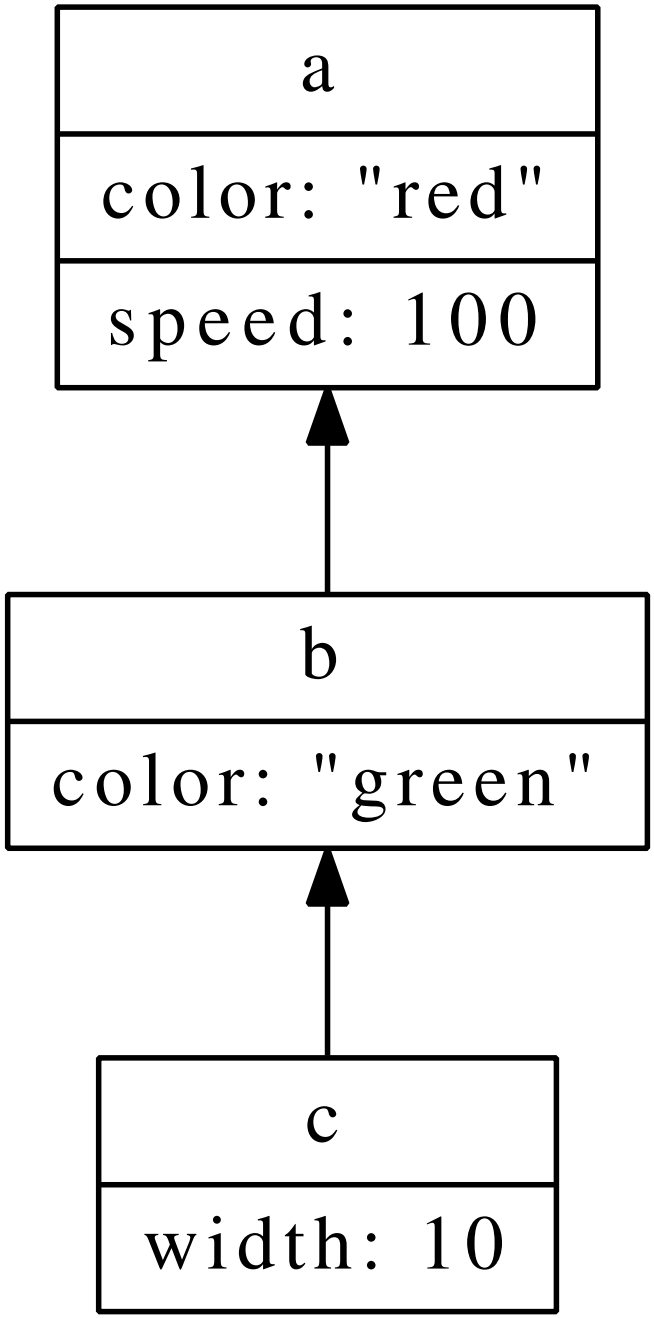
- Resides in the global binding
- Inner functions do not capture parent's `this` (there are several workarounds such as `let self = this;`, `bind`, and ES2015 arrow functions)
- The `this` object can be set manually! (Take a look at the `call`, `apply`, and `bind` functions.)

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

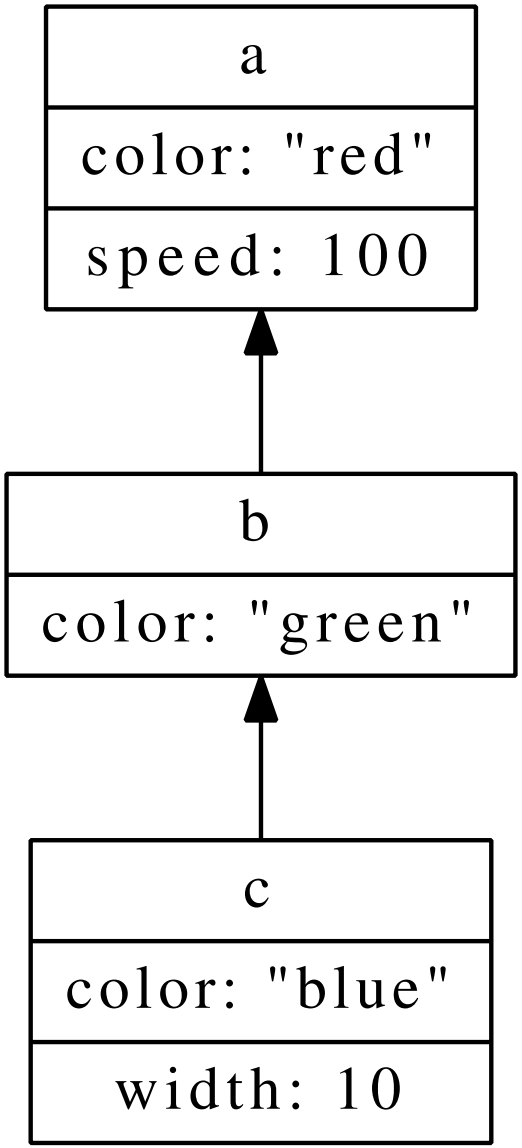



```
c.color === "green";
c.speed === 100;
```

Manual Configuration of Inheritance

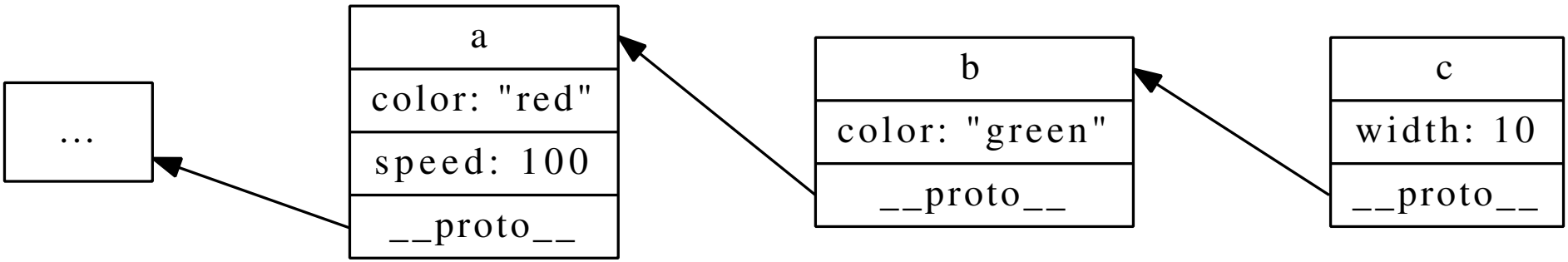
```
let a = {color: "red", speed: 100};
let b = Object.create(a);
let c = Object.create(b);

c.speed; // 100
```

Setting Properties and Inheritance




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

Inheritance with `__proto__`



Prototype Details

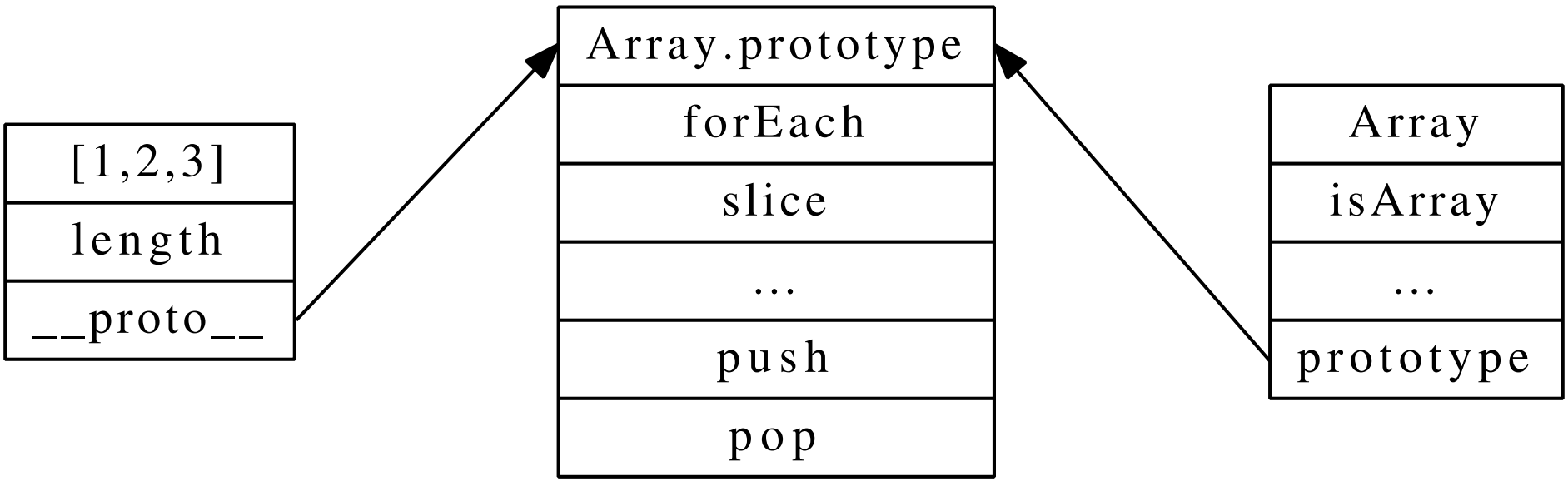
- All objects have an internal link to another object called its *prototype* (known internally as the `__proto__` property).
- The prototype object also has a prototype, and so on up the *prototype chain* (the final link in the chain is `null`).
- Objects *delegate* properties to other objects through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

Using `__proto__` in ES2015

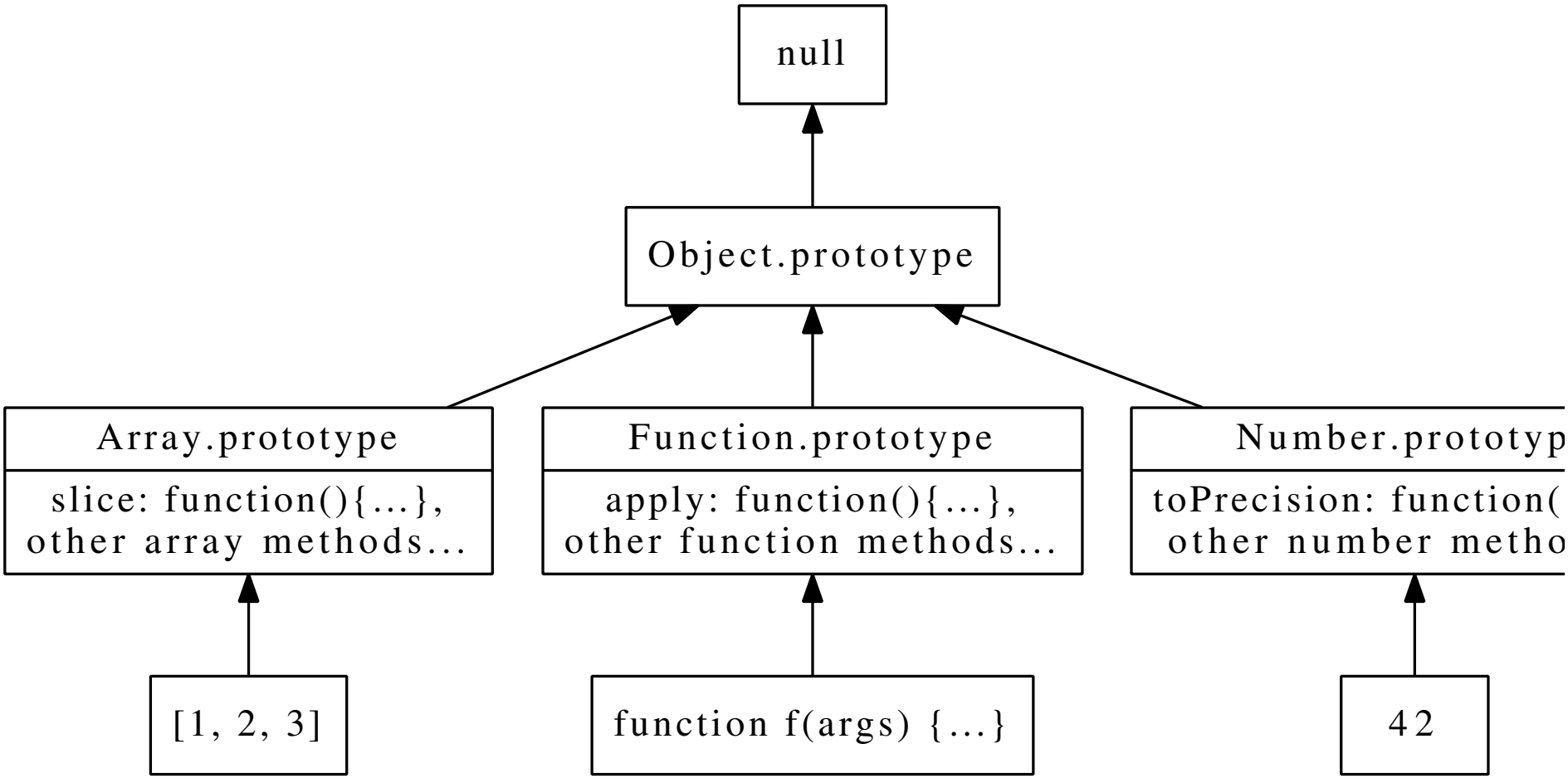
Starting in ECMAScript 2015, the `__proto__` property is standardized as an accessible property.

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

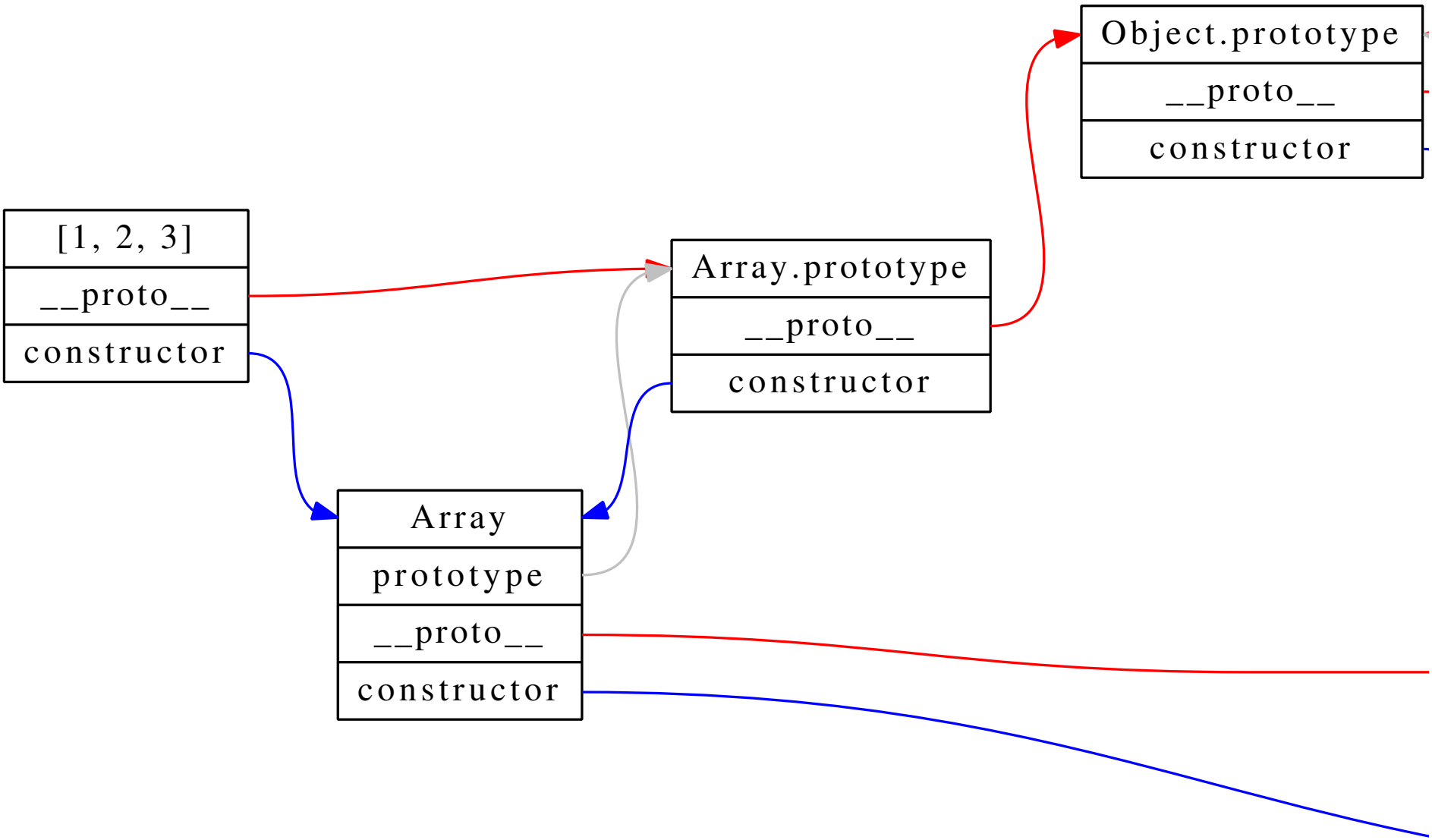
Looking at `Array` Instances



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

```
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

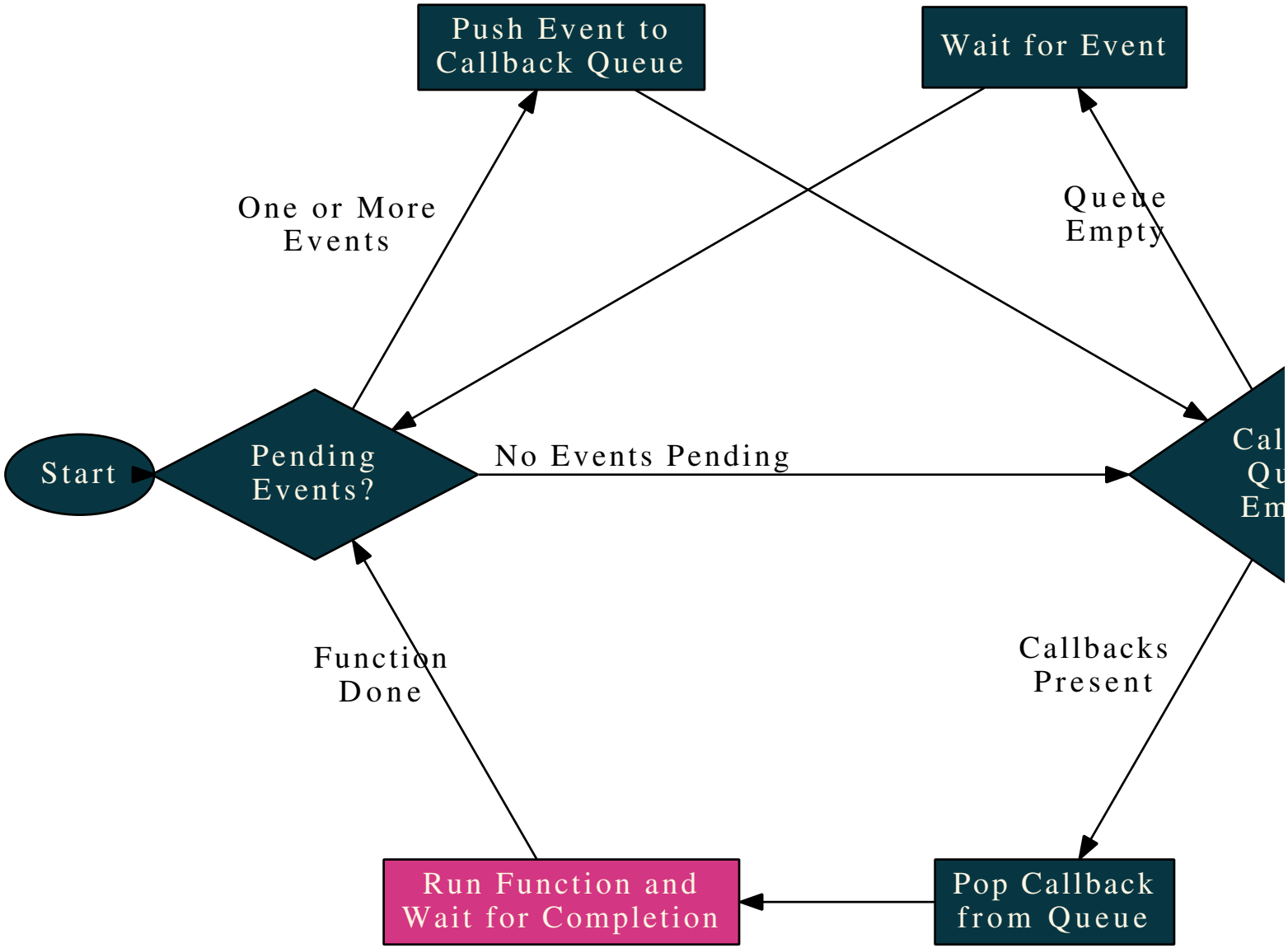
The JavaScript Runtime

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`)

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");
});
```

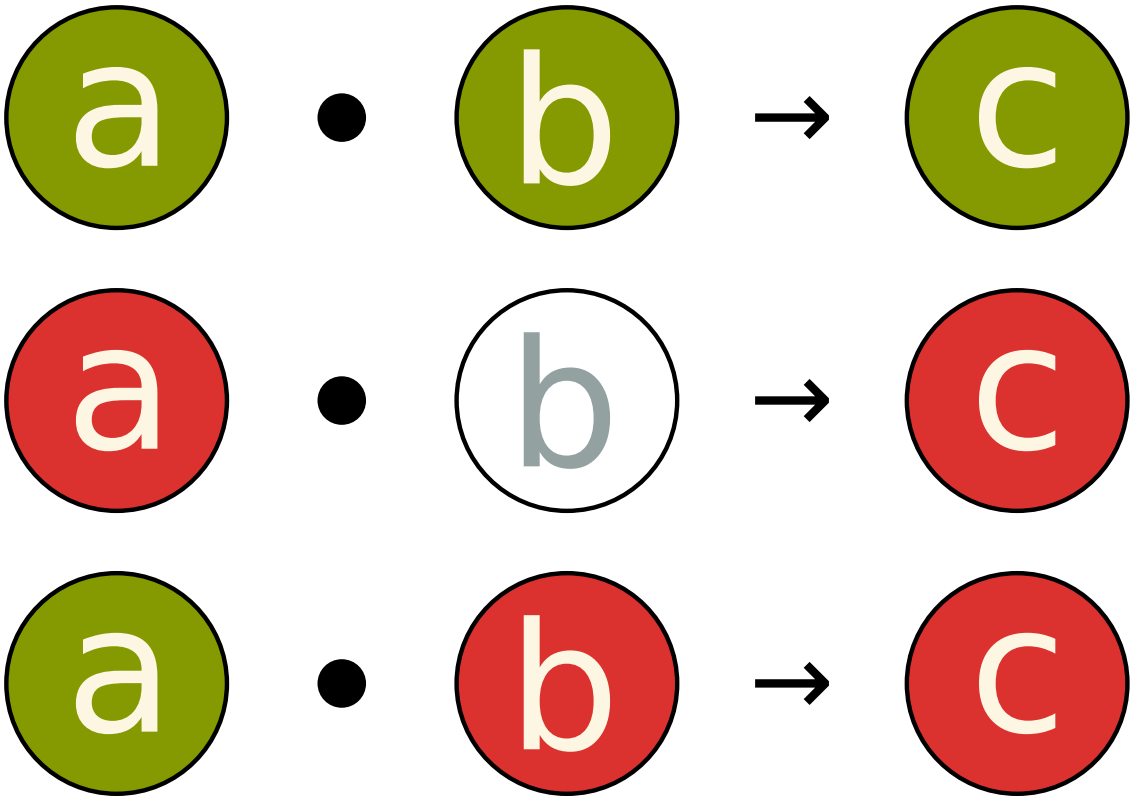
Callbacks Using Promises

```
$.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);
  });
```

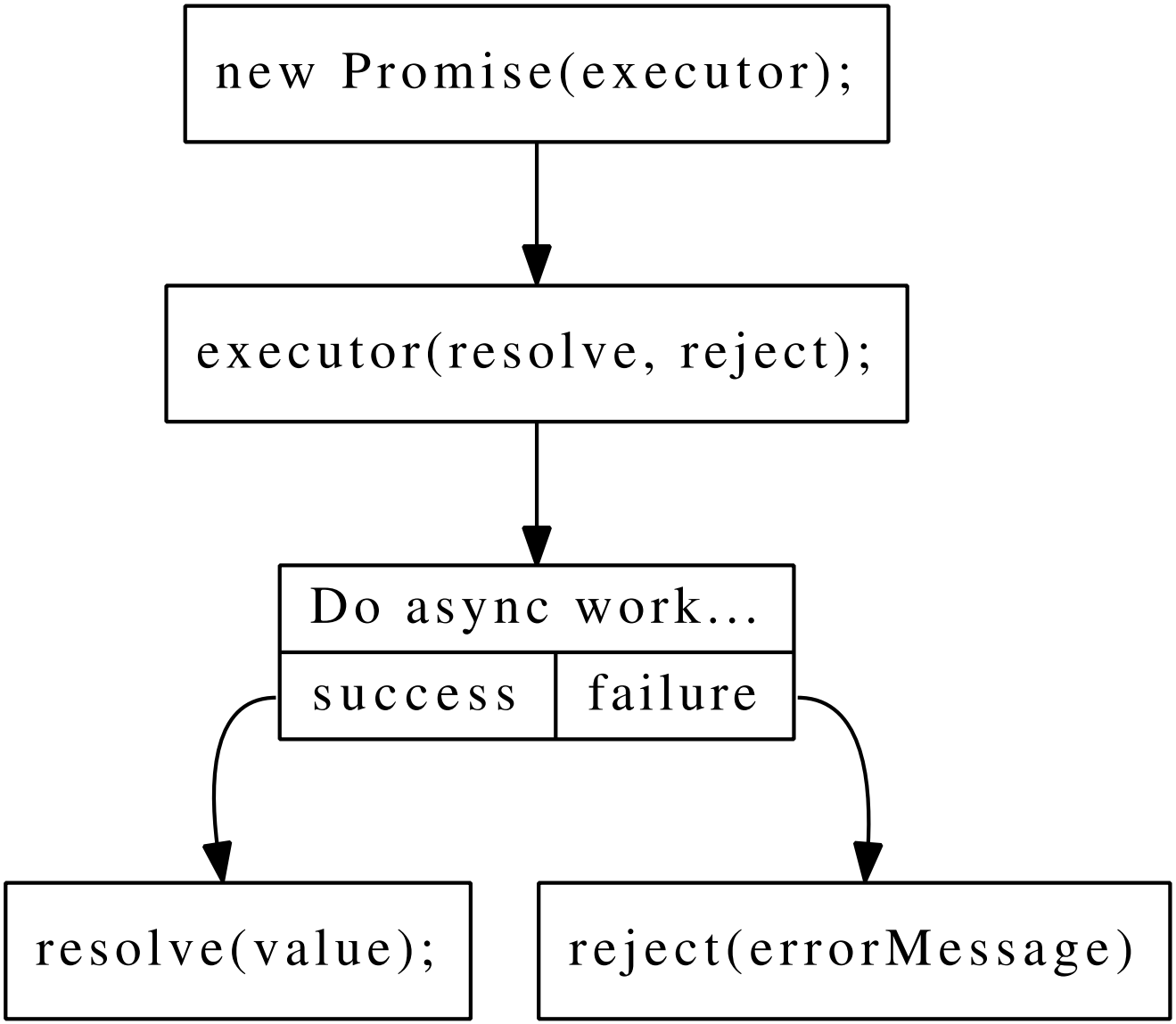
Promise Details

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

Visualizing Promises (Composition)



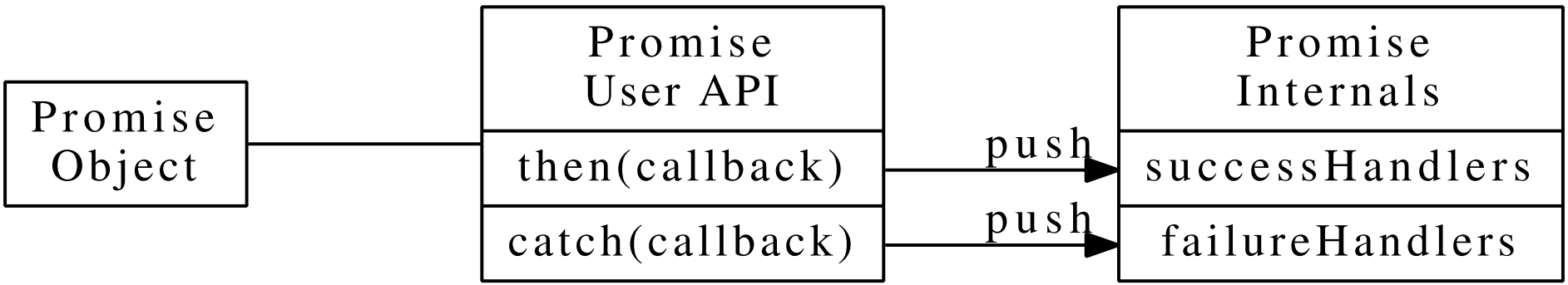
Visualizing Promises (Owner)



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/`

Web Components

The Major Parts of Web Components

Custom Elements Create your own HTML elements
Shadow DOM Give them a private and hidden DOM
Templates Reusable HTML

Custom HTML Elements

The Web Components standard allows us to create custom HTML elements:

- Create an ES2015 class that inherits from `HTMLElement`
- Pick the name for your new HTML element (must contain a hyphen ("-"))
- Register your class as a handler for the custom element name

Autonomous Custom Elements

Create new HTML elements that do whatever you want!

```
class ChatBox extends HTMLElement { }
customElements.define("chat-box", ChatBox);
```

and in your HTML:

```
<chat-box></chat-box>
```

Lifecycle Callbacks

Custom element classes can respond to a small number of events by defining methods:

`constructor`: Element created (don't forget to call `super()`)
`connectedCallback`: The custom element was added to the DOM
`disconnectedCallback`: Removed from the DOM
`attributeChangedCallback`: Notification for observed attributes

Example: Autonomous Custom Element

```
class HelloAutonomous extends HTMLElement {
  constructor() {
    super();
    this.textContent = "Hello World";
  }
}

customElements.define("hello-autonomous", HelloAutonomous);
```

(See: `src/www/js/apis/components/example.js`)

The Shadow DOM

Custom elements can have their own DOM which is private and hidden. It's call the *shadow* DOM.

- A single element may have a complicated DOM behind it (think of the `<video>` element)
- Isolates JavaScript and CSS so only the shadow DOM is affected
- Perfect for encapsulated components!

Example: Creating and Using a Shadow DOM

```
class HelloShadow extends HTMLElement {
  constructor() {
    super();

    const shadowRoot = this.attachShadow({mode: "open"})

    const style = document.createElement("style");
    style.textContent = "p { color: red; }";
    shadowRoot.appendChild(style);

    const p = document.createElement("p");
    p.textContent = "Hello World in red!";
    shadowRoot.appendChild(p);
  }
}

customElements.define("hello-shadow", HelloShadow);
```

HTML Templates

A standard way of dealing with reusable HTML templates:

- The `<template>` element for creating templates
- The `<slot>` element to mark placeholders in templates

Example: HTML Templates

```
<!-- Create a template and slots: -->
<template id="with-name">
  <ul>
    <li>Hello <slot name="first-name">World</slot>!</li>
    <li>Your name came from a slot</li>
  </ul>
</template>

<!-- Custom element that fills in a slot: -->
<hello-template>
  <span slot="first-name">Alice</span>
</hello-template>
```

(See: `src/www/js/apis/components/index.html`)

Example: Custom Elements, Shadow DOM, and Templates

```
class HelloTemplate extends HTMLElement {
  constructor() {
    super();

    const template = document.getElementById("with-name");
    const shadowRoot = this.attachShadow({mode: "open"})

    shadowRoot.appendChild(template.content.cloneNode(true));
  }
}

customElements.define("hello-template", HelloTemplate);
```

(See: `src/www/js/apis/components/example.js`)

Browser Support

- Custom Elements and Templates
  - IE (No support)
  - Edge (No support)
  - Firefox >= 63 (2018)
  - Safari >= 10.1 (2017)
  - Chrome >= 53 (2016)
- Shadow DOM
  - IE (No support)
  - Edge (No support)
  - Firefox >= 63 (2018)
  - Safari >= 11.1 (2018)
  - Chrome >= 66 (2018)

(Polyfills exist for most browsers.)

Exercise: Creating a Web Component

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

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

Exercise: Artist Details with Slots

1. Open the following files:
   - `src/www/js/discography/components/show.js`
   - `src/www/js/discography/index.html`
2. Fill in the missing pieces for exercises 3
3. Play with your web component:

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

Exercise: (Bonus) Listing Albums

1. Open `src/www/js/discography/components/show.js`
2. Fill in the bonus pieces
3. Play with your web component:

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

Decorators

What are Decorators?

Decorators provide an official mechanism in JavaScript for metaprogramming. In other words, they add the ability for run-time code generation.

- Functions that generate code
- Are given an object that fully describes the code from which they were invoked
- Are invoked by using `@` in front of their name, and placed before classes, methods, properties, etc.

Example Decorator

```
function final(descriptor) {
  let { kind } = descriptor;
  console.assert(kind === "class");

  function finisher(klass) {
    Object.freeze(klass);
    Object.freeze(klass.prototype);
  }

  return { ...descriptor, finisher };
}
```

Using the Decorator

```
@final
class Hello {
  say() { console.log("Hello!") };
}
```

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

```
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("v1");
    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)

Observable

Observable Basics

Observables are:

- Sort of like promises, but for multiple values over time
- A functional way of dealing with events (push-based values)
- Another way to embrace functional programming in JavaScript
- Blends functional programming and the Observer Pattern

Example: Subscribing to Events

When subscribing to an Observable you provide a function that will get called each time a value is delivered:

```
const button = document.querySelector("button");
const span   = button.parentNode.querySelector("span");

// `countClicks' is a function that returns an observable:
countClicks(button)
  .subscribe(n => span.textContent = n);
```

(See: `src/www/js/apis/rxjs/example.js`)

Example: Observables from Events

There are many ways to create an Observable. The `fromEvent` function creates an Observable that delivers event objects:

```
function countClicks(element) {
  return fromEvent(element, "click")
    .pipe(
      // Limit to two clicks per second:
      throttleTime(500),

      // A running counter of clicks:
      scan(n => n + 1, 0)
    );
}
```

(See: `src/www/js/apis/rxjs/example.js`)

Browser Support

There is no native browser support for observable (it's still a stage 1 proposal). However, there are several implementations:

- RxJS
- zen-observable
- fate-observable

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

```
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 Bable

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 Bable 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/`

General Testing Overview

Testing in the Browser

In order to achieve comprehensive testing in JavaScript you need to:

- Test your code in the web browser
- Then test it in every browser you support
- And use a tool that automates this process

The Two Major Flavors of Testing

- Assertion-based testing:

```
assert("empty objects", objects.length > 0);
```

- Expectation-based testing:

```
expect(objects.length).toBeGreaterThan(0);
```

Behavior-driven Development with Jasmine

What is Jasmine?

- Specification-based testing
- Expectations instead of assertions
- Provides the testing framework
- Only provides a very simple way to run tests

Example: Writing Jasmine Tests

```
describe("ES2015 String Methods", function() {
  describe("Prototype Methods", function() {
    it("has a find method", function() {
      expect("foo".find).toBeDefined();
    });
  });
});
```

Basic Expectation Matchers

`toBe(x)`: Compares with `x` using `===`.
`toMatch(/hello/)`: Tests against regular expressions or strings.
`toBeDefined()`: Confirms expectation is not `undefined`.
`toBeUndefined()`: Opposite of `toBeDefined()`.
`toBeNull()`: Confirms expectation is `null`.
`toBeTruthy()`: Should be true `true` when cast to a Boolean.
`toBeFalsy()`: Should be `false` when cast to a Boolean.

Numeric Expectation Matchers

`toBeLessThan(n)`: Should be less than `n`.
`toBeGreaterThan(n)`: Should be greater than `n`.
`toBeCloseTo(e, p)`: Difference within `p` places of precision.

Smart Expectation Matchers

`toEqual(x)`: Can test object and array equality.
`toContain(x)`: Expect an array to contain `x` as an element.

Exercise: Writing a Test with Jasmine

1. Open `src/www/js/jasmine/adder.spec.js`
2. Read the code then do exercise 1 (we'll do exercise 2 later)
3. To test and debug, open

   `src/www/js/jasmine/index.html`

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.

Deferred (Pending) Tests

Tests can be marked as pending either by:

```
it("declared without a body!");
```

or:

```
it("uses the pending function", function() {
  expect(0).toBe(1);
  pending("this isn't working yet!");
});
```

Spying on a Function or Callback (Setup)

```
let foo;

beforeEach(function() {
  foo = {
    plusOne: function(n) { return n + 1; },
  };
});
```

Spying on a Function or Callback (Call Counting)

```
it("should be called", function() {
  spyOn(foo, 'plusOne');
  let x = foo.plusOne(42);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(foo.plusOne).toHaveBeenCalledTimes(1);
  expect(foo.plusOne).toHaveBeenCalledWith(42);

  expect(x).toBeUndefined();
});
```

Spying on a Function or Callback (Call Through)

```
it("should call through and execute", function() {
  spyOn(foo, 'plusOne').and.callThrough();
  let x = foo.plusOne(42);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(x).toBe(43);
});
```

Spying on a Function or Callback (Call Fake)

```
it("should call a fake implementation", function() {
  spyOn(foo, 'plusOne').and.callFake(n => n + 2);
  let x = foo.plusOne(42);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(x).toBe(44);
});
```

Exercise: Using Jasmine Spies

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

   `src/www/js/jasmine/index.html`

Testing Time-Based Logic (The Setup)

```
let timedFunction;

beforeEach(function() {
  timedFunction = jasmine.createSpy("timedFunction");
  jasmine.clock().install();
});

afterEach(function() {
  jasmine.clock().uninstall();
});
```

Testing Time-Based Logic (setTimeout)

```
it("function that uses setTimeout", function() {
  inFiveSeconds(timedFunction);

  // The callback shouldn't have been called yet:
  expect(timedFunction).not.toHaveBeenCalled();

  // Move the clock forward and trigger timeout:
  jasmine.clock().tick(5001);

  // Now it's been called:
  expect(timedFunction).toHaveBeenCalled();
});
```

Testing Time-Based Logic (setInterval)

```
it("function that uses setInterval", function() {
  everyFiveSeconds(timedFunction);

  // The callback shouldn't have been called yet:
  expect(timedFunction).not.toHaveBeenCalled();

  // Move the clock forward a bunch of times:
  for (let i=0; i<10; ++i) jasmine.clock().tick(5001);

  // It should have been called 10 times:
  expect(timedFunction.calls.count()).toEqual(10);
});
```

Testing Asynchronous Functions

```
describe("asynchronous function testing", function() {
  it("uses an asynchronous function", function(done) {

    // `setTimeout' returns immediately,
    // so this test does too!
    setTimeout(function() {
      expect(done instanceof Function).toBeTruthy();
      done(); // tell Jasmine we were called.
    }, 1000);

  });
});
```

Exercise: Using Jasmine Spies

1. Open `src/www/js/jasmine/delayed.spec.js`
2. Read the code then do exercise 3
3. To test and debug, open

   `src/www/js/jasmine/index.html`

Running Jasmine Tests

- Standalone runner:
  - List files in `SpecRunner.html`
  - Opening that file in your browser runs the tests
- Node.js runner:
  - Provides a `jasmine` tool
  - Runs tests inside Node.js
- Karma-Jasmine runner:
  - Automatically manages browser farms
  - Runs tests in parallel on all browsers
  - Can use headless browsers (PhantomJS)
  - Support for continuous integration

Best Practices for Testing

- Make sure your tests actually fail
- Separate pure logic from DOM manipulation
- Test with valid *and* invalid input (or use fuzzing)
- Automate your tests so they run all the time
- Avoid mocking/spies if you can (they create "holes")

Further Information

See the following for more information:

- Jasmine documentation
- Karma test runner

Other testing frameworks:

- JSPec: Full-featured behavior testing
- Sinon: Spies, stubs, and mocks
- Chai: Testing assertion library

Browser Automated Testing

End-to-End Testing Options