Intermediate JavaScript

A One Day Learning Spike

April 3, 2018

Peter J. Jones

✉ pjones@devalot.com

@devalot

http://devalot.com

DEVALOT

What's In Store

Before Lunch	After Lunch
Quick Review	Asynchronous Programming
Advanced Functions	Testing w/ Jasmine
Object-Oriented Programming	Browser APIs

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.log(e);
  }

  var bar = function(f) {
    var c = 2;
    a = 12;
    return a + c + b;
  };
}
```

Loops and Closures

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

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.)

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.

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*.

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

```
var 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`.

HTML and Text Content

```
var 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.

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: Warming Up with the DOM and Events

1. Open the following files:
   - `src/www/js/warmup/warmup.js`
   - `src/www/js/warmup/index.html` (read only!)
2. Open the `index.html` file in your web browser
3. Follow the instructions in the JavaScript file

Hint: Use MDN as an API reference.

Modules, Namespaces, and Packages

- Organize logical units of functionality
- Prevent namespace clutter and collisions
- Several options for module implementation
  - The module pattern
  - CommonJS modules
  - ECMAScript 6th Edition modules

The Module Pattern

- Allows for private methods and functions
- Useful for creating namespaces
- Uses an anonymous closure to hide private functionality and make a public interface

Immediately-Invoked Function Expressions: Basics

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

Immediately-Invoked Function Expressions: Expanded

```
(function() {  // (1) Anonymous function expression.

  var x = 1; // (2) Body of function.
  return x;

})();  // (3) Close function and call function.
```

Example: Module Pattern

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

  // Private method.
  var 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

The arguments Variable

- Array-like interface. But not exactly an array:

```
arguments.length;   // Some number.
arguments[0];       // First argument.
arguments.forEach; // undefined :(
```

Converting `arguments` into an Array

Converting the `arguments` property into an array isn't as straight forward as it should be. The following code is a common idiom:

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

or, with ES6:

```
var 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
```

Function.prototype.call

Calling a function and explicitly setting `this`:

```
var x = {color: "red"};
var 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:

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

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

var 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:

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

x.f = f;

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

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

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

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

var 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:

```
var obj = {
  magnitude: 10,

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

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

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

What's Wrong with This Code?

Assuming this function is called millions of times:

```
var digitName = function(n) {
  var names = ["zero", "one", "two", /* more elements */];
  return names[n] || "";
};
```

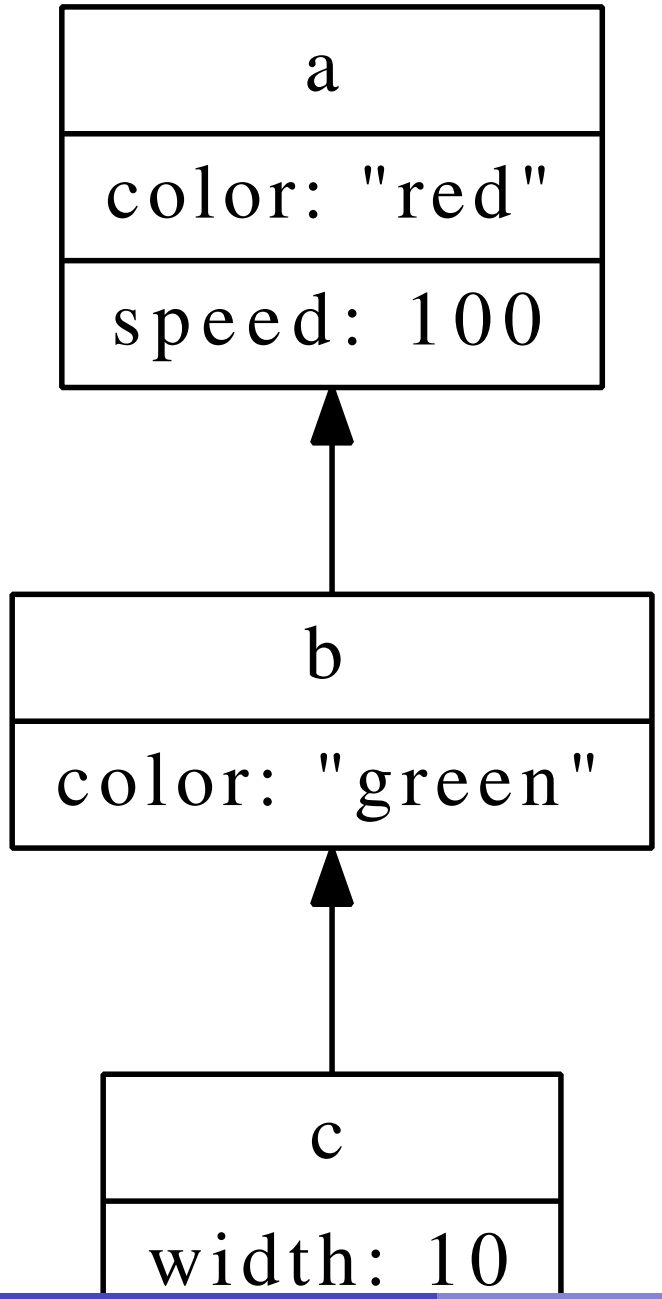
Lazy Function Definitions to the Rescue

```
var digitName = function(n) {
  var names = ["zero", "one", "two", /* more elements */];

  // No `var' here!
  digitName = function(n) {
    return names[n] || "";
  };

  return digitName(n);
};
```

Inheritance in JavaScript

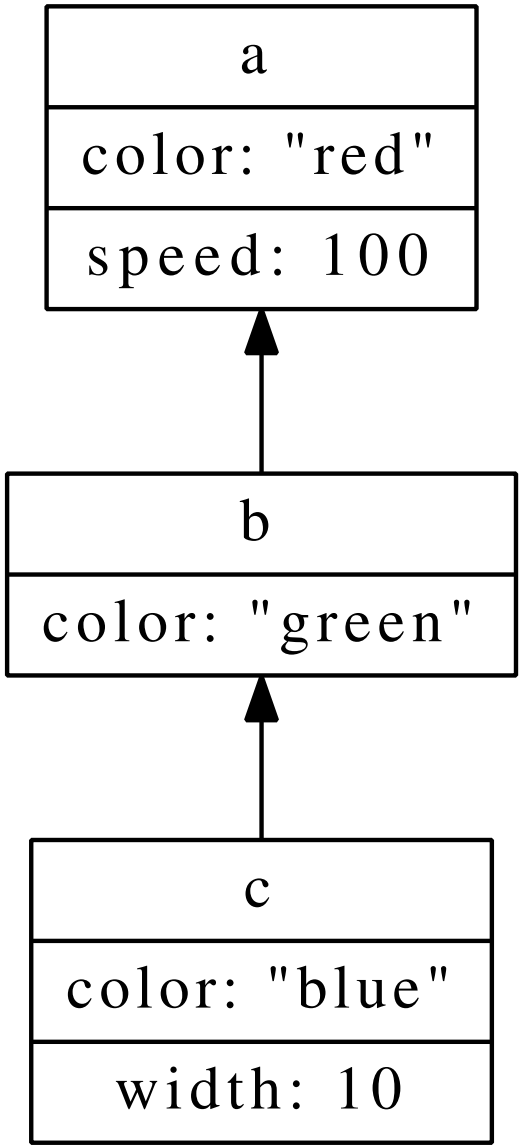
- JavaScript doesn't use classes, it uses prototypes
- There are ways to simulate classes (even ES6 does it!)
- The prototypal model:
  - Tends to be smaller
  - Less redundant
  - Can simulate classical inheritance as needed
  - More powerful

Object Inheritance




```
c.color === "green";
c.speed === 100;
c.width === 10;
```

Object Inheritance




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

Figure 2:Setting a Property

Prototype Refresher

- 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.

Inheritance with `__proto__`

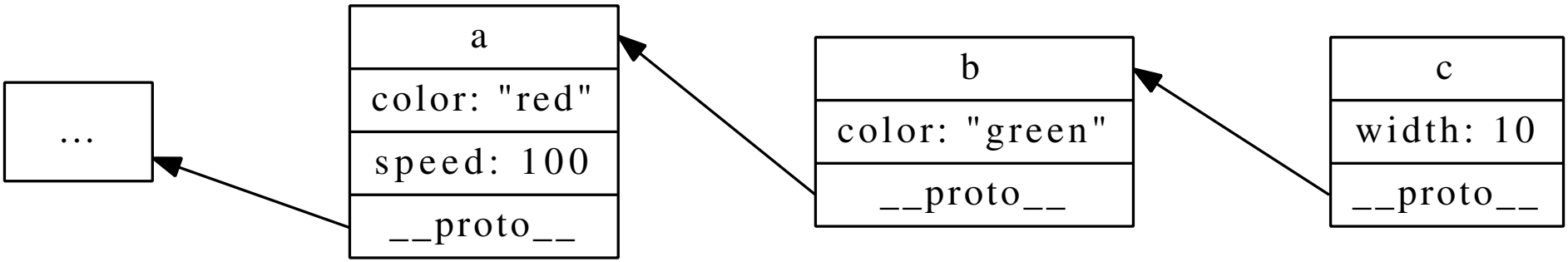


Figure 3:Prototypes

Looking at `Array` Instances

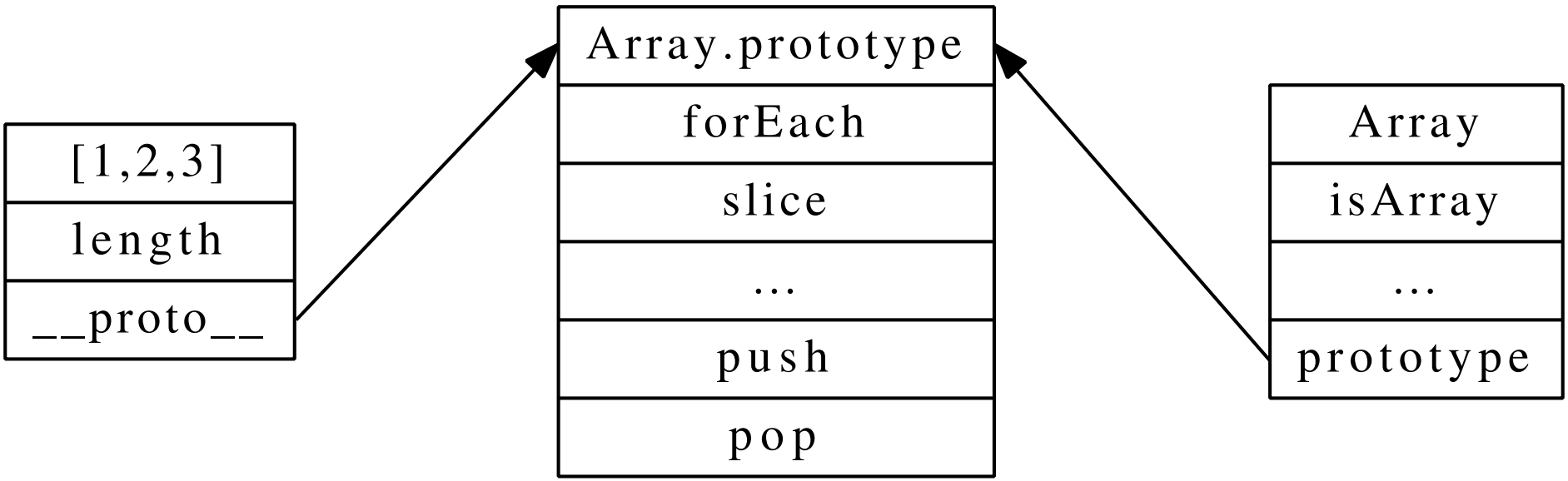


Figure 4:Array and Array.prototype

The Prototype Chain

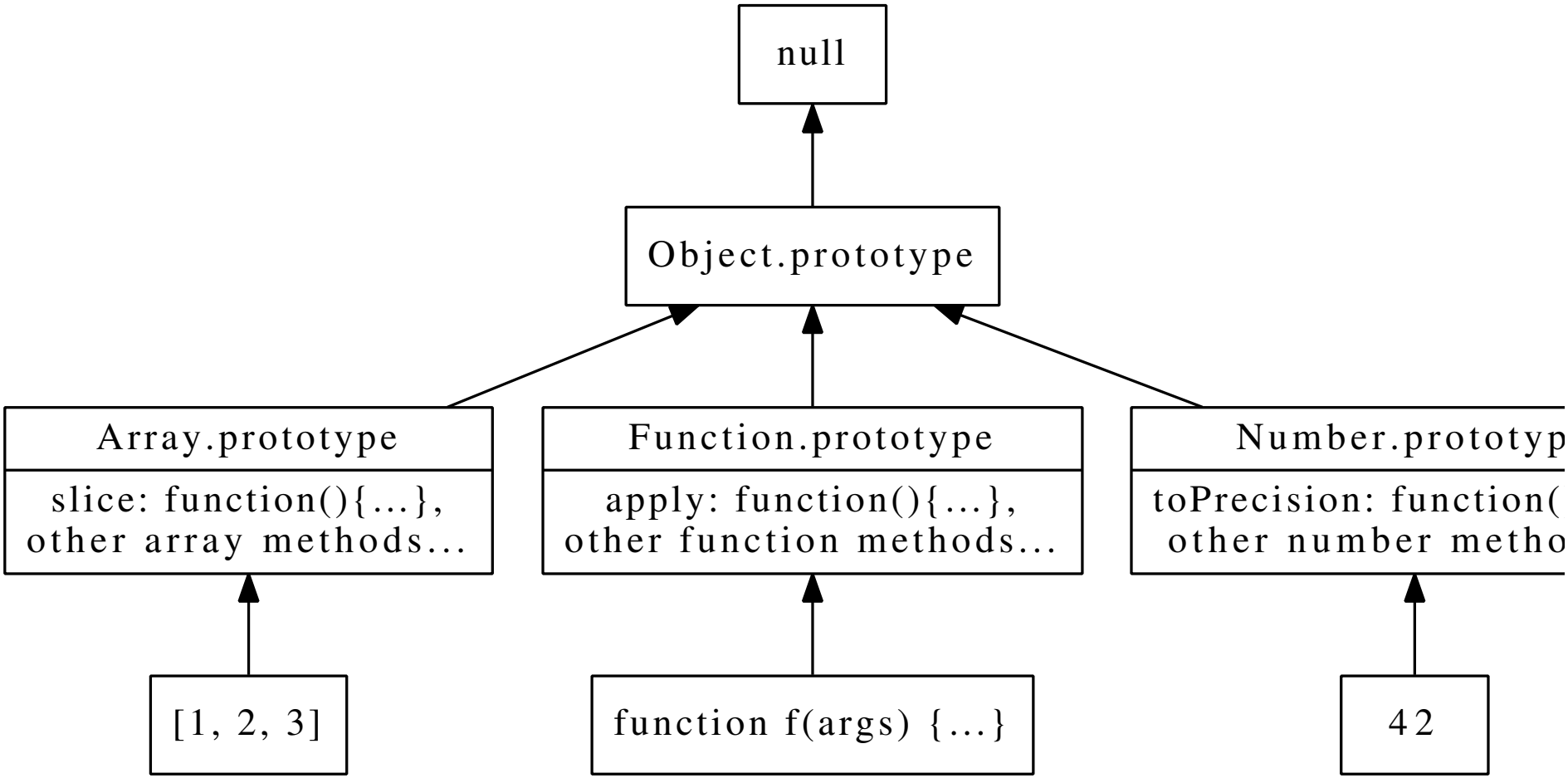


Figure 5:Prototypal Inheritance

Another Look at `Array` Instances

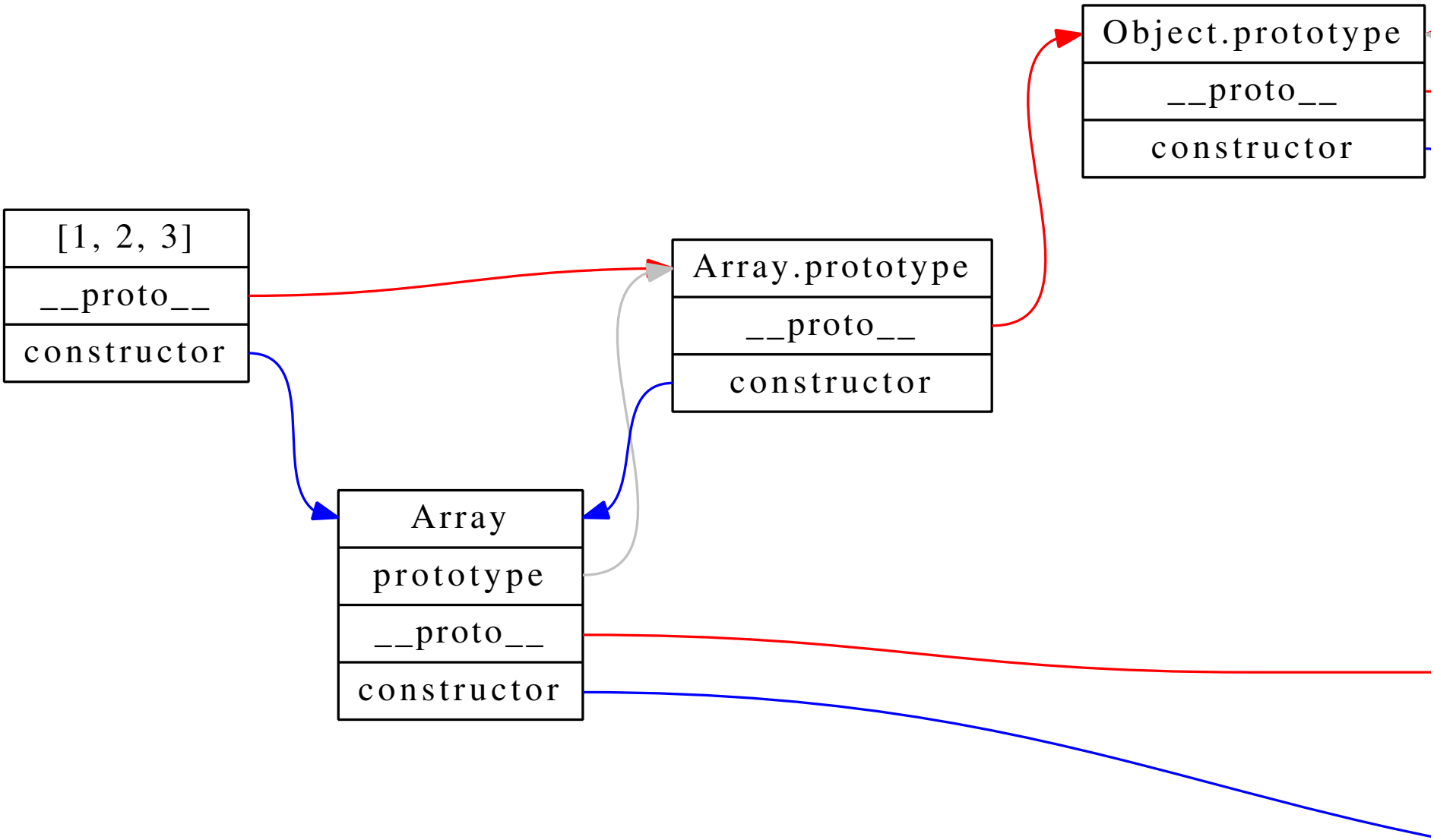


Figure 6:Array and Friends

Using `Object.create`

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

```
var x = Object.create(Array.prototype);
x.push(1);
```

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.

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

// Is like:

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

Constructor Functions and OOP

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

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

var rect = new Rectangle(10, 20);
console.log(rect.area());
```

Constructor Functions and Inheritance

```
var Square = function(width) {
  Rectangle.call(this, width, width);
  this.isSquare = true;
};

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

var sq = new Square(10);
console.log(sq.area());
```

Using `__proto__` in ES6

Starting in ECMAScript Edition 6, the `__proto__` property is standardized as an accessible property.

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

Exercise: Class Builder

1. Open the following files:
   - `src/www/js/builder/builder.spec.js` (read only!)
   - `src/www/js/builder/builder.js`
2. Implement the `Builder` function:
   It should generate a constructor function using the `constructor` property given to it. The remaining properties become prototype properties.
3. Use the `index.html` file to run the tests

Constructors that Aren't

Parasitic inheritance is created by:

- Constructor or factory functions
- They don't create their own objects
- After having another function create an object they augment it in some way.

An Example Using the new Operator

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

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

var Square = function(width) {
  var rect = new Rectangle(width, width);
  rect.isSquare = true;
  return rect;
};

var sq = new Square(10);
console.log(sq.area());
```

What is a Mixin?

- Simulates multiple inheritance
- Properties from interesting objects are copied into the target object, making the target object appear to be made up of the interesting objects.
- All the same problems you get with real multiple inheritance, but without any of the built-in solutions to resolve them.

Using the Mixin Technique

```
var A = function() {};
A.prototype.isA = function() {return true};

var B = function() {};
B.prototype.isB = function() {return true};

var C = function() {};
C.prototype.isC = function() {return true};

C.mixin(A, B);
var obj = new C();

console.log(obj.isA()); // true
console.log(obj.isB()); // true
console.log(obj.isC()); // true
```

Writing the Mixin Machinery

```
Function.prototype.mixin = function() {
  var i, prop;

  for (i=0; i<arguments.length; ++i) {
    for (prop in arguments[i].prototype) {
      this.prototype[prop] =
        arguments[i].prototype[prop];
    }
  }
};
```

Simple Introspection Techniques

- The `instanceof` Operator:

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

- The `isPrototypeOf` Function:

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

- The `Object.getPrototypeOf` Function:

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

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

Object.defineProperty

```
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 (Boolean)
  - `value`: The property's value
  - `writable`: If the value can change (Boolean)

Introduction to Debugging

- All modern browsers have built-in JavaScript debuggers
- We've been using the debugging console the entire time!

Browser Debugging with the Console

- The `console` object:
  - Typically on `window` (doesn't always exist)
  - Methods
    - `log`, `info`, `warn`, and `error`
    - `table(object)`
    - `group(name)` and `groupEnd()`
    - `assert(boolean, message)`

Accessing the Debugger

- In the browser's debugging window, choose **Sources**
- You should be able to see JavaScript files used for the current site

Setting Breakpoints

There are a few ways to create breakpoints:

- Open the source file in the browser and click a line number
- Right-click the line number to create conditional breakpoints
- Use the `debugger;` statement in your code

Stepping Through Code

- After setting breakpoints, you can reload the page (or trigger a function)
- Once the debugger stops on a breakpoint you can step through the code using the buttons in the debugger
  - Step In: Jump into the current function call and debug it
  - Step Over: Jump over the current function call
  - Step Out: Jump out of the current function

Console Tricks

- `$_` the value of the last evaluation
- `$0`—`$4` last inspected elements in historical order
- `$("selector")` returns first matching node (CSS selector)
- `$$("selector")` returns all matching nodes
- `debug(function)` sets a breakpoint in `function`
- `monitor(function)` trace calls to `function`

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

- Unit tests:

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

- Specification tests:

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

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("ES6 String 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)`: `Math.abs(e - actual) < (Math.pow(10, -p) / 2)`

Smart Expectation Matchers

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

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)

```
var 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');
  var x = foo.plusOne(1);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(x).toBeUndefined();
});
```

Spying on a Function or Callback (Call Through)

```
it("should call through and execute", function() {
  spyOn(foo, 'plusOne').and.callThrough();
  var x = foo.plusOne(1);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(x).toBe(2);
});
```

Testing Time-Based Logic (The Setup)

```
var 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 (var 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() {
      done(); // tell Jasmine we were called.
    }, 1000);

  });
});
```

Running Jasmine Tests

- [Standalone][jasmine-standalone] runner:
  - List files in `SpecRunner.html`
  - Opening that file in your browser runs the tests
- [Node.js runner][jasmine-npm]:
  - Provides a `jasmine` tool
  - Runs tests inside Node.js
- [Karma-Jasmine][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

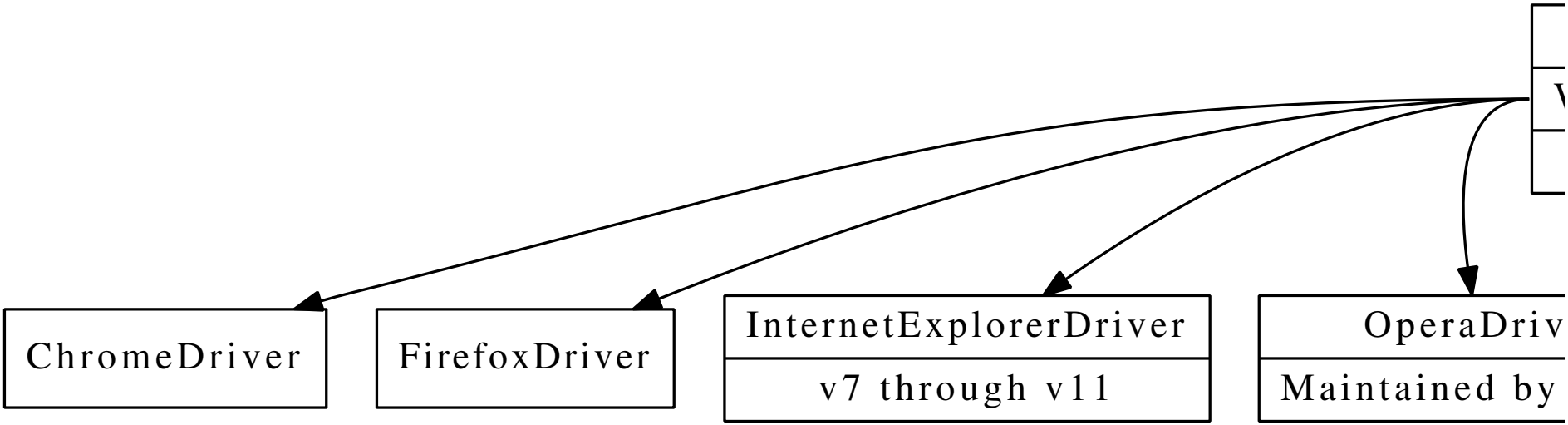
See the following for more information:

- [Jasmine][] documentation
- [Karma][] test runner

Other testing frameworks:

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

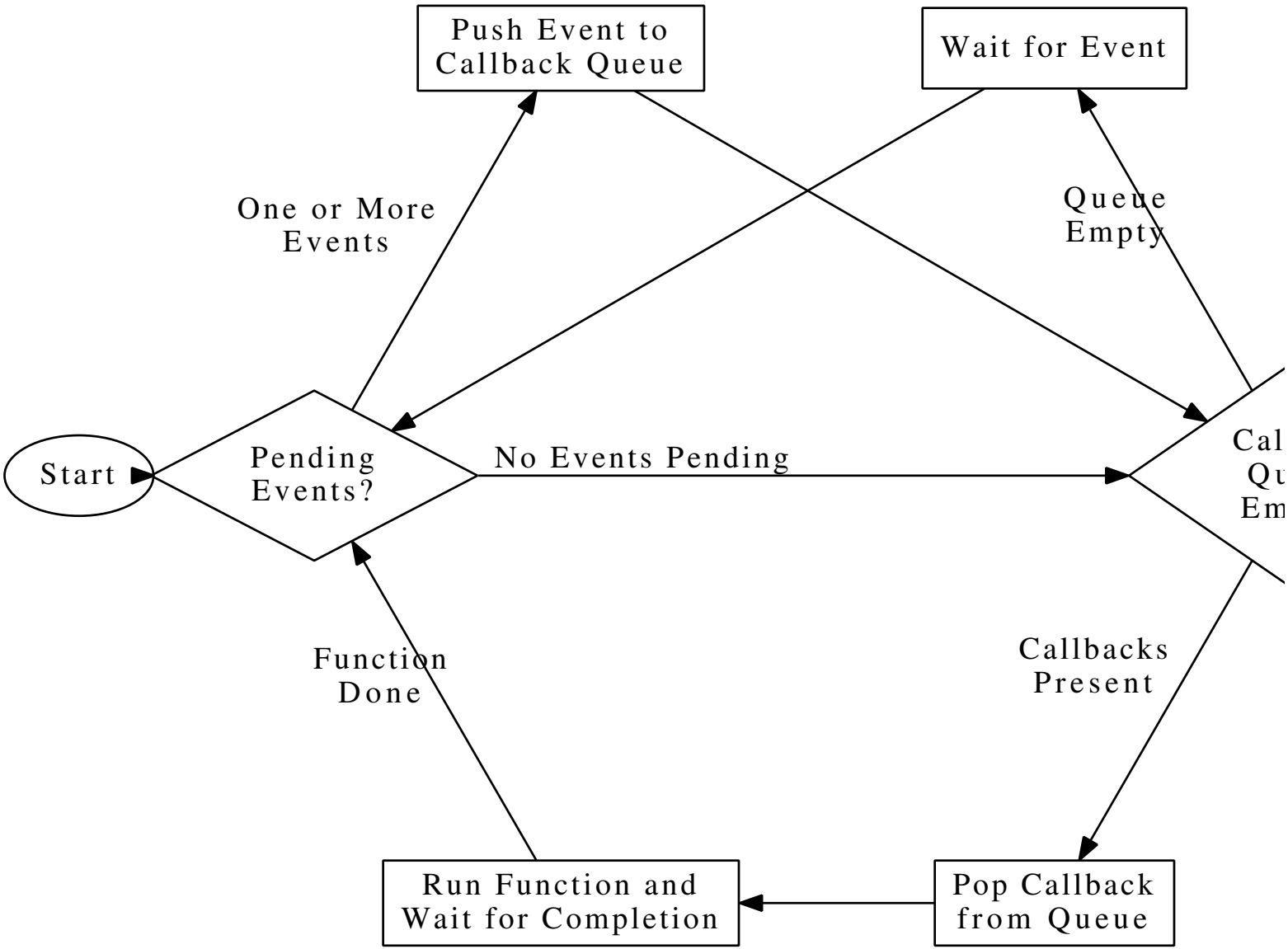
End-to-End Testing Options



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

Visualizing the Runtime



Callbacks without Promises

```
$.get("/a", function(data_a) {
  $.get("/b/" + data_a.id, function(data_b) {
    $.get("/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");
});
```

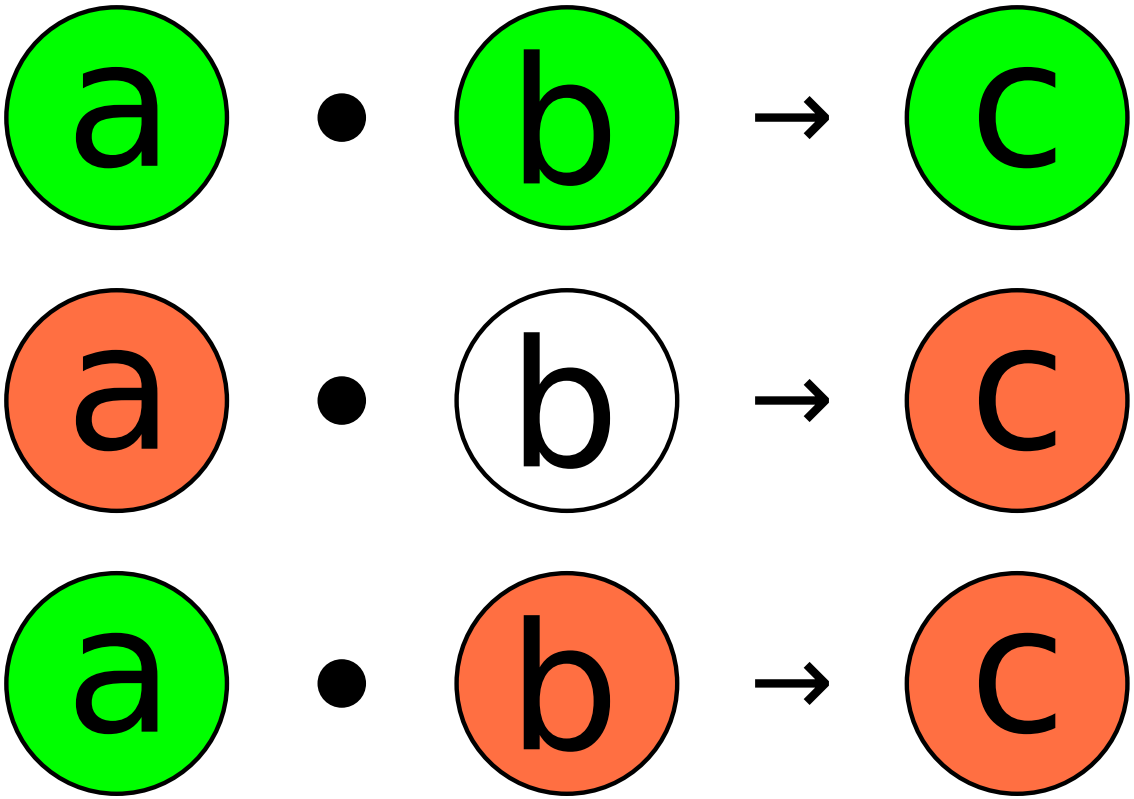
Callbacks Using Promises

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

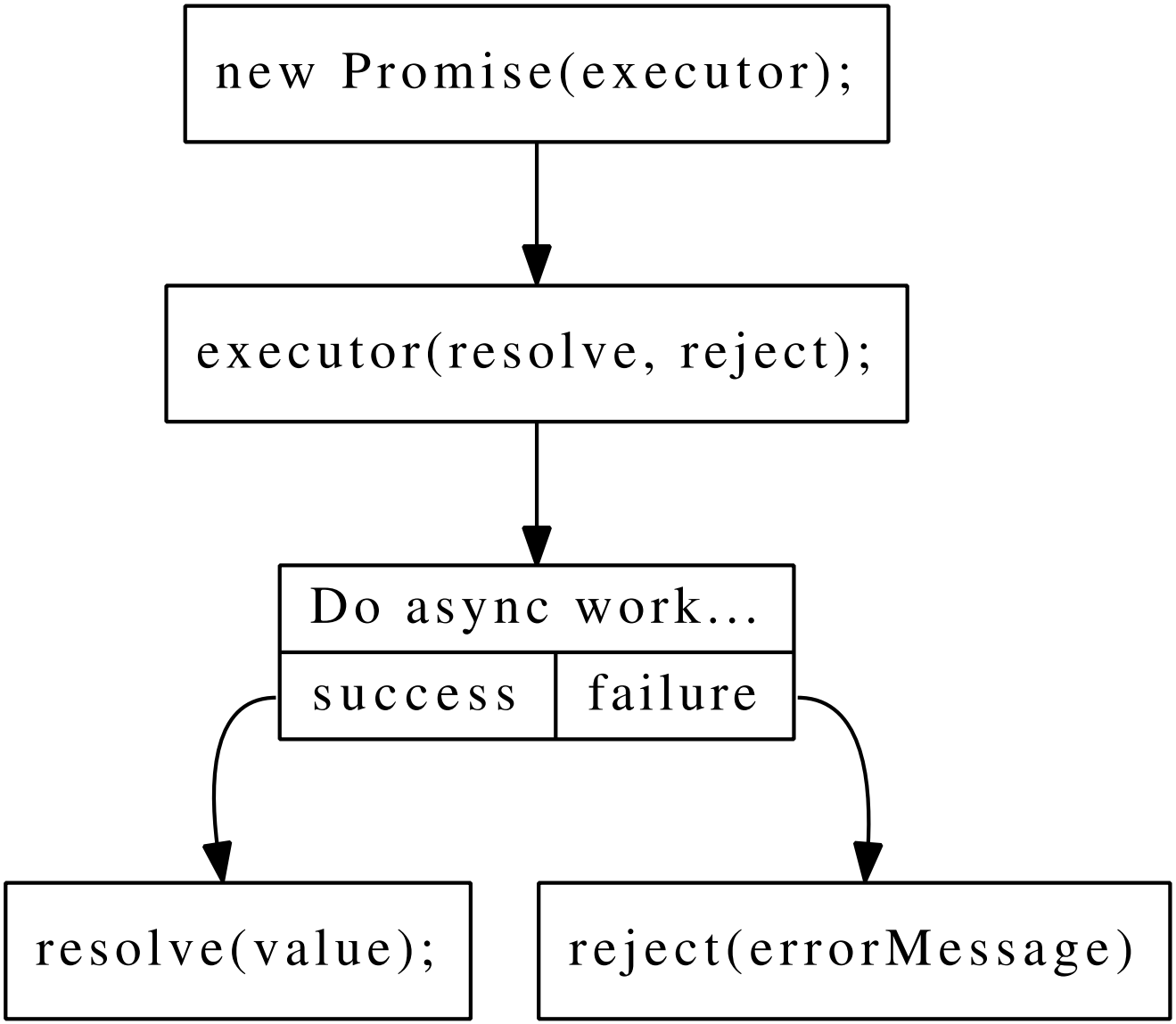
Promise Details

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

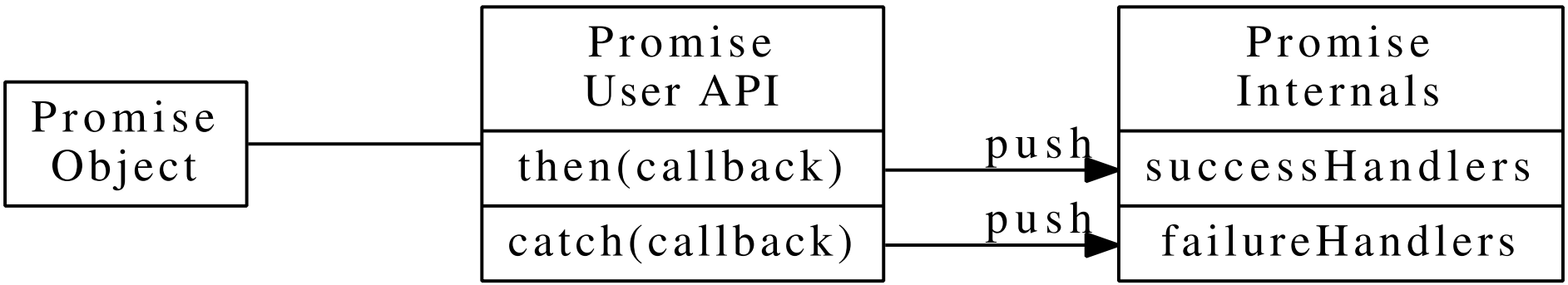
Visualizing Promises (Composition)



Visualizing Promises (Owner)



Visualizing Promises (User)



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();
});
```

Ajax Refresher

Making an Ajax request:

```
var req = new XMLHttpRequest();

req.addEventListener("load", function(e) {
  if (req.status == 200) {
    console.log(req.responseText);
  }
});

req.open("GET", "/example/foo.json");
req.send(null);
```

Exercise: A Simple Ajax Library

1. Open `src/www/js/ajax/ajax.js`
2. Fill in the missing pieces
3. Open the `index.html` file in your browser
4. Get the tests in `index.html` to pass

Exercise: Using Your Ajax Library

1. Open `src/www/js/artists/artists.js`
2. Complete the exercise using your Ajax library
3. Open the `index.html` file in your browser
4. Play with your code!

The New `let` Keyword

- ES6 introduces `let`
- Declare a variable in the scope of containing block:

```
if (expression) {
  var a = 1; // scoped to wrapping function
  let b = 2; // scoped to the block
} // Woah!
```

Hoisting and `let`

It does not hoist!

```
{
  console.log(b); // Error!

  let b = 12;
  console.log(b); // No problem.
}
```

Looping with `let`

Using let with a for loop is possible in ES6:

```
for (let i=0; i<10; i++) {
  // i is bound to a new scope each iteration
  // getting its value reassigned
  // at the end of the iteration
}
```

Preventing Reassignment

The `const` keyword defines a block-level variable that must be initialized when it's declared and can't be reassigned:

```
var f = function() {
  const x = "foo";

  // ...

  x = 1;  // Ignored.
};
```

Arrow Functions

```
element.addEventListener("click", function(e) {
  // ...
});

// Becomes:


element.addEventListener("click", e => {
  // ...
});
```

Implicit `return` for Arrow Expressions

If you omit curly braces you can write a single expression that automatically becomes the return value of the function:

```
a.map(function(e) {
  return e + 1;
});

// Becomes:

a.map(e => e + 1);
```

Arrow Warnings

- Arrow function do not have a `this` or an `arguments` variable!
- If you use curly braces you need to use `return`.

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.)

Array Destructuring

```
let firstPrimes = function() {
  return [2, 3, 5, 7];
};

let x, y, rest;
[x, y, ...rest] = firstPrimes();

console.log(x); // 2
console.log(y); // 3
console.log(rest); // [ 5, 7 ]
```

- Similar to *pattern matching* from functional languages
- The *lvalue* can be an array of names to bind from the *rvalue*

(Object destructuring is part of ES2018.)

Classes

New `class` keyword that provides syntactic sugar over prototypal inheritance:

```
class Square extends Rectangle {
  constructor(width) {
    super(width, width);
  }
  someMethod() {
    return "Interesting";
  }
}
```

Class Features

- Class statements are *not* hoisted.
- Classes can also be defined using an expression syntax:

```
var Person = class {
  // ..
};
```

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.)

The Object.assign Function

Copies properties from one object to another:

```
var o1 = {a: 1, b: 2, c: 3};
var o2 = { };

Object.assign(o2, o1);
console.log(o2);
```

Produces this output:

```
{ a: 1, b: 2, c: 3 }
```

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

Modules

- 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();
```

New Generic `for` Loop

The new `for...of` loop can work with any object that supports iteration:

```
var anything = [1, 2, 3];

for (let x of anything) {
  console.log(x);
}
```

Generators

```
let something = {
  [Symbol.iterator]: function*() {
    for (let i=0; i<10; ++i) {
      yield i;
    }
  },
};

for (let x of something) {
  console.log(x);
}
```

Iterators

```
let something = {
  [Symbol.iterator]: function() {
    let n = 0;

    return {
      next: () => ({value: n, done: n++ >= 10}),
    };
  },
};

for (let x of something) {
  console.log(x);
}
```

Maps

```
let characters = new Map();

characters.set("Ripley", "Alien");
characters.set("Watney", "The Martian");

characters.has("Ripley"); // true
characters.get("Ripley"); // "Alien"
```

WeakMaps

- Like a `Map`, but *keys* can be garbage collected
- Similar API as a `Map` (missing some functions)
  - `WeakMap.prototype.delete`
  - `WeakMap.prototype.get`
  - `WeakMap.prototype.set`
  - `WeakMap.prototype.has`

Others

- `Set` and `WeekSet`
  Mathematical sets, as well as a weak version.
- `Proxy` and `Reflect`
  Powerful objects for metaprogramming.
- `Symbol`
  Create and use runtime unique entries in the symbol table.
- Template Literals
  String interpolation:

```
`Hello ${name}`
```

Exponentiation Operator

Prior to ES7:

```
Math.pow(4, 2);
```

New in ES7:

```
4 ** 2;
```

Array.prototype.includes

A new prototype function to test if a value is in an array.

Prior to ES7:

```
[1, 2, 3].indexOf(3) >= 0;
```

New in ES7:

```
[1, 2, 3].includes(3);
```

Async Functions

Major improvement to asynchronous functions thanks to promises and generators. Asynchronous callbacks are hidden with new syntax.

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

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

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

Summary of Other Changes

- String padding (ensuring a string is the proper length)
  - `String.prototype.padStart`
  - `String.prototype.padEnd`
- `Object.values` and `Object.entries`
- `Object.getOwnPropertyDescriptors`
- Trailing commas in function parameters and call arguments
- Shared memory (`SharedArrayBuffer`)
- Atomic operations (e.g., `Atomics.store`)

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");
var 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");
var 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

Documentation

- https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage
- https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

What is the AppCache?

- A server-side manifest file
- Tells the browser which files to long-term cache
- Allows a web site to work offline

Example Manifest File

Add a `manifest` attribute to your HTML:

```
<html manifest="/site.appcache">
  <!-- ... -->
</html>
```

Create the manifest file on your server:

```
CACHE MANIFEST

CACHE:
/favicon.ico
index.html
app.js
app.css

NETWORK:
*
```

Server-side Requirements

- The server must transmit the manifest file with the `Content-Type` set to `text/cache-manifest`
- The server should send the correct cache and `E-Tag` headers to the browser to keep the browser from caching the manifest file too long
- The manifest file should be generated server-side with comments in the file containing the `E-Tag` headers for each listed file

Client-side Considerations

- Once you start using application caching the cache becomes the default source for *all* requests
- The browser will use the application cache even if the user is online
- The browser won't allow network traffic back to the site for uncached resources by default
- Make sure your manifest has a `NETWORK:` section with `*`

Updating the Cache in Long-lived Applications

1. Periodically (once a day) call `update`:

```
applicationCache.update();
```

2. Listen for update events and notify the user:

```
(function(cache) {
  cache.addEventListener('updateready', function() {
    if (cache.status === cache.UPDATEREADY) {
      // Tell the user to reload the page.
    }
  });
})(applicationCache);
```

Browser Support

- IE >= 10
- Firefox >= 3.5
- Safari >= 4
- Chrome >= 4
- Opera >= 11.5

Further Reading

- A Beginner's Guide to Using the Application Cache
- Offline Web Applications (Spec)

Canvas: Two Drawing APIs

- 2D drawing primitives via paths
- 3D drawing via WebGL
- Both can be hardware accelerated
- Typically 60 FPS (if animating)

Drawing a Circle: The HTML

```
<canvas id="circle"></canvas>
```

Drawing a Circle: JavaScript

```
canvas  = document.getElementById("circle");
context = canvas.getContext("2d");

var path = new Path2D();
path.arc(75, 75, 50, 0, Math.PI * 2, true);
context.stroke(path);
```

Browser Support

- IE >= 9
- Firefox >= 1.5
- Safari >= 2
- Chrome >= 1
- Opera >= 9

Documentation

https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial

What the File API Is, and Isn't

- It's *not* a general-purpose I/O interface
- It only lets you get basic info about user-selected files:
  - Name
  - Size
  - MIME type
- A user selects a file with an `<input>` or using drag and drop

Example: Chosen File Size

- In the HTML:

```
<input type="file" id="the-input"/>
```

- In the JavaScript (after the user picks a file):

```
var input = document.getElementById("the-input");
var size = input.files[0].size;
```

Browser Support

- IE >= 10
- Firefox >= 3.0
- Safari >= 6.0
- Chrome >= 13
- Opera >= 11.5

Documentation

`https://developer.mozilla.org/en-US/docs/Web/API/File`

Testing If Geolocation is Enabled

```
if ("geolocation" in navigator) {
  // ...
}
```

Getting the Browser's Location

```
navigator.geolocation.getCurrentPosition(function(pos) {
  // ...
});
```

Browser Support

- IE >= 9
- Firefox >= 3.5
- Safari >= 5
- Chrome >= 5
- Opera >= 16

Documentation

https://developer.mozilla.org/en-US/docs/Web/API/Geolocation/Using_geolocation

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);
  });
```

Browser Support and Documentation

Browsers:

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

Docs:

- Living Standard
- MDN

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

Documentation

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

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

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

Documentation and Demos

- MDN: WebSockets API
- MDN: WebSockets Example
- socket.io: Popular Library

A Word About Server-Sent Events

- Pros:
  - Simpler than WebSockets
  - One direction: server to browser
  - Uses HTTP, no need for a custom protocol
- Cons:
  - Not supported in IE (any version)
  - Poor browser support in general (polyfills are available)
- How:
  - Browser: use the `EventSource` global object
  - Server: just write messages to the HTTP connection
- Docs:
  - See MDN

Languages that Compile to JavaScript

- PureScript
- Flow
- TypeScript
- Dart

PureScript

- Purely functional programming language that compiles to JS
- Strong, static type system (similar to Haskell)
- Clean, human-readable JavaScript output
- Lots of open source modules for PureScript

Flow

- Language extension to JavaScript
- Standalone static type checking system
- Runs as part of your build process
- Uses Babel to transpile to standard JavaScript
- Sponsored by Facebook

Flow Features

- Type inference (no type annotations required)
- Syntax for type annotations so you can be explicit
- Automatic `null` checking
- Enabled per-file or per-function

What Does it Look Like?

Adding types to a function:

```
// Explicit type annotations:
var add = function(x: number, y: number): number {
  return x + y;
};

// This will fail type checking:
add("1", 2);

// Also fails type checking:
var sum = add(1, 2);
console.log(sum.length);
```

Using Flow

1. Allow Flow to process a file by adding a comment flag:

   ```
   // @flow
   ```

2. Type check the code by running `flow check`
3. Use Babel to remove the type annotations

Flow Demo Application

1. `http://localhost:3000/alternatives/flow/`
2. `www/alternatives/flow`
3. Before it will work you need to:

```
$ npm install -g gulp-cli
$ npm install
$ gulp
```

TypeScript

- A language based on ES6 (classes, arrow functions, etc.)
- All features compile to ES5
- Same basic type-annotation syntax as Flow
- Type inference and null-checking are weaker than Flow
- Sponsored by Microsoft

Dart

- OOP Language standardized as ECMA-408
- Optional type system
- Requires a runtime system in JavaScript
- Sponsored by Google

Popular ES6 to ES5 Transpilers

- Babel
- Traceur

Looking to the Future

- WebAssembly