

# Targeted Topics

Peter J. Jones

✉ [pjones@devalot.com](mailto:pjones@devalot.com)

🐦 [@devalot](https://twitter.com/devalot)

<http://devalot.com>



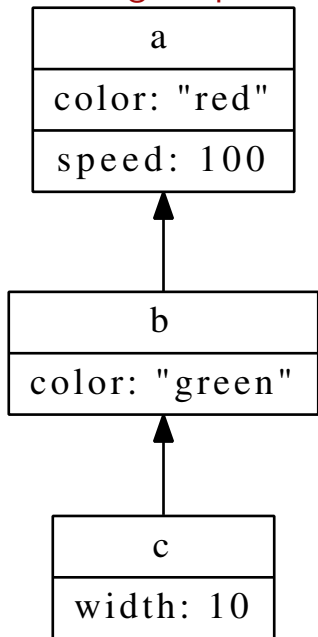
DEVALOT

# JavaScript Classes and the Prototype

# 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();
```

## Inheriting Properties from Other Objects

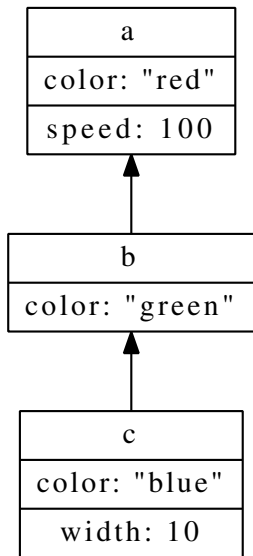


```
c.color === "green";  
c.speed === 100;
```

# Manual Configuration of Inheritance

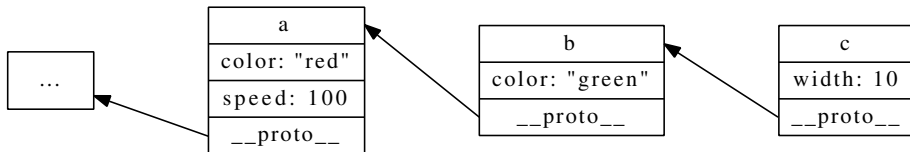
```
let a = {color: "red", speed: 100};  
let b = Object.create(a);  
let c = Object.create(b);  
  
c.speed; // 100
```

# Setting Properties and Inheritance

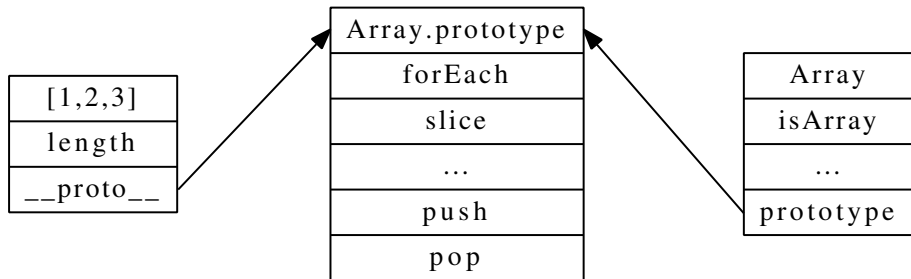


```
c.color = "blue";  
c.color === "blue";
```

# Inheritance with `__proto__`



# Looking at Array Instances





# 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 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);  
    }  
  });  
}
```

# 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

## 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



# Functional Programming with JavaScript

# 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 known 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 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

# 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*.)

## 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
```

# 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!");
```



## 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.

# 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);},  
  };  
})();
```

# Introducing Higher-order Functions

The `forEach` function is a good example of a *higher-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;  
});
```

## 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>)

# 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>)



## 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);
```

# 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

*-- 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`

# What is a *Pure* Function?

Pure functions are functions that have the same properties as their mathematical cousins. Some of these properties include:

- Can only access bound variables (i.e., their arguments)
- Cannot have side effects (e.g., update the DOM)
- Given the same inputs, always produces the same output

In other words, a pure function always produces a return value and that return value can only be calculated using the function's arguments.



# What's the Point of Pure Functions?

Pure functions make programming *a lot* easier!

- Everything you need to know about a function is there in its definition
- They don't rely on the state of the program, user, or machine
- Simple to test (can even be automated)

# Writing Pure Functions in JavaScript

Like everything in JavaScript, you get little help from the language when trying to write pure functions. Here are some tips:

- Don't access global or closure variables
- Don't mutate any arguments or call functions that mutate arguments
- Don't change the state of the program or runtime

## Pure Function Quiz: Part 1

```
let checkUserPermission = function(code, roles, cache) {  
  if (cache.includes(code)) {  
    return true;  
  } else if (Object.values(roles).includes(code)) {  
    cache.push(code);  
    return true;  
  }  
  
  return false;  
};  
  
let cache = [];  
let roles = {view: 1, edit: 2, remove: 3};  
  
if (checkUserPermission(3, roles, cache)) {  
  console.log("user can remove page");  
}
```

## Pure Function Quiz: Part 2

```
let emailMatches = function(email, f) {  
  return f(email.subject) || f(email.body);  
};
```

```
let email = {subject: "Foo", body: "Bar"};
```

```
bool = emailMatches(email, function(str) {  
  return str.match(/oo/);  
});
```

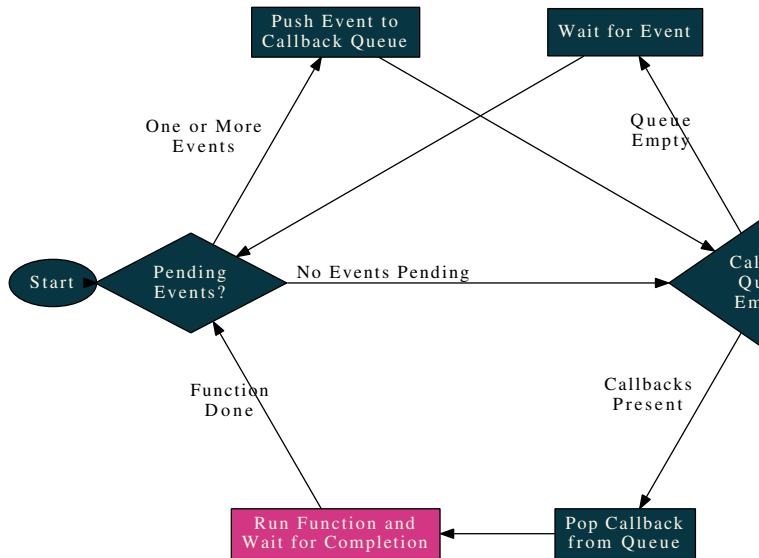
# Asynchronous Programming

# Introduction to the Runtime

- JavaScript has a single-threaded runtime
- Work is therefore split up into small chunks (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>)

## 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");  
});
```



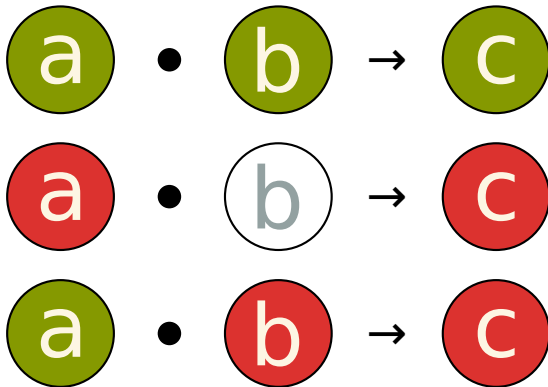
# Callbacks Using Promises

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

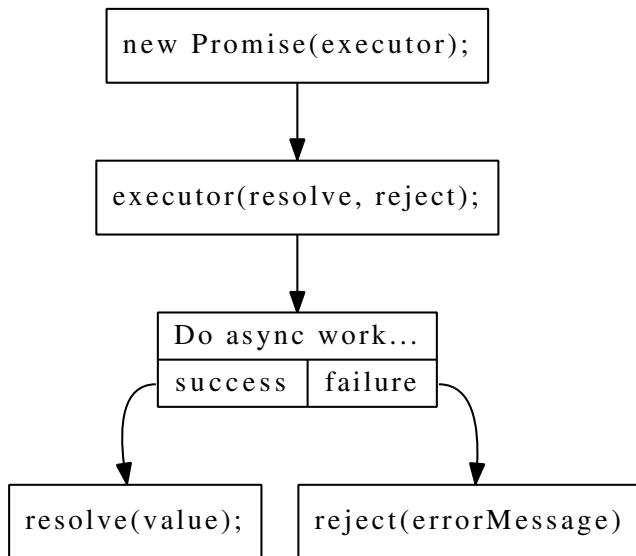
# Promise Details

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

# Visualizing Promises (Composition)



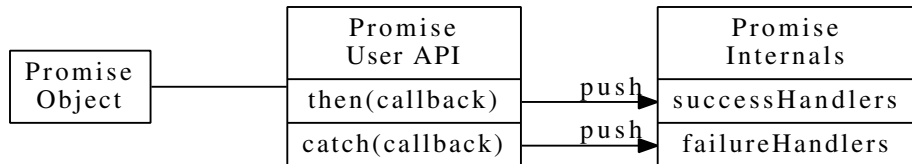
# Visualizing Promises (Owner)



## 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();  
});
```

## 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  $\geq 14$
- Firefox  $\geq 34$
- Safari  $\geq 10.1$
- Chrome  $\geq 42$
- Opera  $\geq 29$

# 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/`

# 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/`

## Modern JavaScript: ES2015 - ES2018

# ES2015 Summary

- New keywords: `let`, `const`, `class`, `import`, `export`, etc.
- New function syntax (i.e. arrow functions)
- New syntax for function parameters
- New syntax for destructuring
- New built-in objects
- Lots more

# The New `let` Keyword

- ES2015 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 ES2015:

```
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:

```
let 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:

```
let 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.)

# 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

# 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 WeakSet  
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}``

# ES2016 Summary

- New operator: `**`
- New function: `Array.prototype.includes`

# Exponentiation Operator

Prior to ES2016:

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

New in ES2016:

```
4 ** 2;
```

## Array.prototype.includes

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

Prior to ES2016:

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

New in ES2016:

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

# ES2017 Summary

- Async functions!!
- Updates to the String object
- Small changes to Object.prototype
- A few others

# Async Functions

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

```
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`.  
  }  
}
```



# 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`)

# ES2018 Summary

- Rest and spread operations for properties (proposal)
- New function: `Promise.prototype.finally` (proposal)
- Asynchronous iterators and generators (proposal)
- Regular expression improvements (s flag, groups, lookbehind, unicode)
- Template literal improvements (proposal)

# Object Destructuring and Rest Property Assignment

```
let x = {a: 1, b: 2, c: 3, d: 4};  
let {a, b, ...z} = x;
```

```
a; // 1
```

```
b; // 2
```

```
z; // { c: 3, d: 4 }
```

# Object Initialization Spreading

```
let z = {c: 3, d: 4};  
let x = {a: 1, b: 2, ...z};  
  
x; // { a: 1, b: 2, c: 3, d: 4 }
```

## Promise.prototype.finally

The `finally` function allows you to respond to a promise being resolved or rejected. It's perfect for updating the UI after a network call finishes:

```
startSpinner();

$getJSON("/foo")
  .finally(() => stopSpinner())
  .then(data => updateUI(data));
```

# Asynchronous Iterators for JavaScript

- async iterator and generator functions that yield promises
- await version of the for-of loop

```
for await (const line of readLines(filePath)) {  
  console.log(line);  
}
```

# Regular Expressions: New Engine Flag

The new `s` engine flag turns on the *dot all* mode:

```
"foo\nbar".match(/foo.bar/); // null
```

```
"foo\nbar".match(/foo.bar/s); // Array(...)
```

# Regular Expressions: Named Capture Groups

Regular expressions can now have named capture groups:

```
m = "2018-06-26".match(/^(?<year>\d{4})-/);  
m.groups.year; // "2018"
```



# Regular Expressions: Lookbehind Assertions

Regular expressions can have lookbehind assertions:

*// Positive Lookbehind:*

```
m = "$9.99".match(/(?<=\$)\d+\.\d+/);  
m[0]; // "9.99"
```

*// Negative Lookbehind:*

```
m = "A1B2C3".match(/(?<![BC])[BC]/);  
m[0]; // "C"
```

# Regular Expressions: Unicode Property Matching

Match Unicode *properties* such as *script*:

```
// U+3C0 is the Greek pi character  
"\u03c0".match(/\p{Script=Greek}/u);
```

(Note the new u engine flag.)

# Template Literals and Escape Sequences

Tagged template literals may contain invalid escape sequences. Those sequences are reserved and made available in a `raw` property:

```
function foo(strings) {  
  strings[0];      // undefined  
  strings.raw[0];  // "I like \u"  
  
  return strings.raw[0].replace("\\u", "you");  
}  
  
foo`I like \u`; // "I like you"
```

# 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!") };  
}
```

# 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`)

## Important Browser APIs

# 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  $\geq 14$
- Firefox  $\geq 34$
- Safari  $\geq 10.1$
- Chrome  $\geq 42$
- Opera  $\geq 29$

# 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/`



# 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  $\geq 63$  (2018)
  - Safari  $\geq 10.1$  (2017)
  - Chrome  $\geq 53$  (2016)
- Shadow DOM
  - IE (No support)
  - Edge (No support)
  - Firefox  $\geq 63$  (2018)
  - Safari  $\geq 11.1$  (2018)
  - Chrome  $\geq 66$  (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/`

# 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  $\geq 10$
- Firefox  $\geq 6$
- Safari  $\geq 6$
- Chrome  $\geq 14$
- Opera  $\geq 12.10$



## 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/`

# 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  $\geq 8$
- Firefox  $\geq 2$
- Safari  $\geq 4$
- Chrome  $\geq 4$
- Opera  $\geq 10.50$

## 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/`

# 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  $\geq 10$
- Firefox  $\geq 3.5$
- Safari  $\geq 4$
- Chrome  $\geq 4$
- Opera  $\geq 10.6$

# 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  $\geq 17$  (2015)
- Firefox  $\geq 44.0$  (2016)
- Safari  $\geq 11.1$  (2018)
- Chrome  $\geq 40$  (2015)
- Opera  $\geq 27$  (2015)

# JavaScript Development Tools

# 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 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

# Introduction to Babel

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

## Manually Using Babel

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

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

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

# Integrating Babel with Your Build Tools

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

Simple overview of a build process:

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



# 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 known 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/`

# 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` 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

# 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);
```