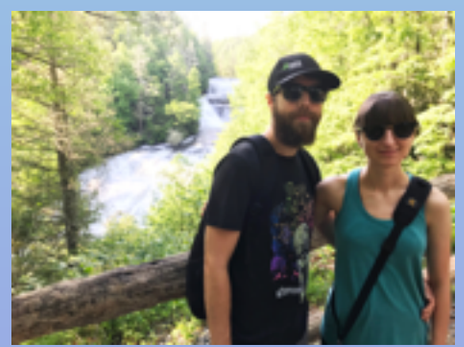
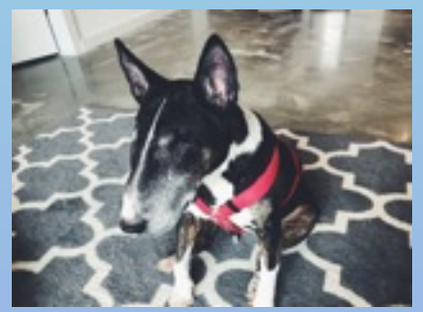
Introduction to ES6

Asheville JavaScript Meetup

About Me

@melindamgolden

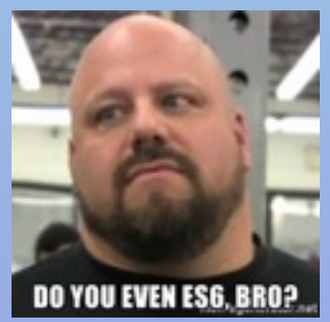




- Been a web developer for over two years
- Work for web design company in Atlanta, work remotely in Asheville
- Self-taught
- Degree in art
- Was a wedding photographer with husband
- Has a bull terrier named Cypress

What is ES6?

- JavaScript is a scripting language that conforms to the ECMAScript specification
- ES6 is the sixth edition of ECMAScript that was released in 2015
- Also known as ES2015 or ECMAScript 2015



- ECMAScript is scripting language specification
- **JavaScript is a scripting language that conforms to the ECMAScript specification**
- **ES6 is the sixth edition of ECMAScript that was released in 2015**
- **Also known as ES2015 or ECMAScript 2015**

Why Should I Use ES6?

- ES6 features are supported across the latest versions of major browsers (http://kangax.github.io/compat-table/es6/)
- Babel converts ES6 into ES5 for older browsers (https://babeljs.io/)
- Introduces new features that make your code more concise and readable
- Addresses some of the 'quirks' of JavaScript

Goodbye Var, Hello Let & Const

- Var variables are **function** scoped
- Var variables can be **updated** or **redefined**

```
function saySomething(string) {
    var intro = 'Hello, '
    console.log(intro + string); // Hello, world
}

saySomething('world');
console.log(intro); // intro is not defined
```

```
var score = 500;
score = 600;
console.log(score); // 600


var name = 'Jeff';
var name = 'Sarah';
console.log(name); // Sarah
```

- To understand let and const, you must first understand var
- **Var variables are function scoped**
- **Var variables can be updated or redefined**
- In the function saySomething(), we have a var variable called intro
- If we try console.log() intro outside of the function, it will say intro is not defined
- Since var is function scoped, it stays inside the function
- In the next example, the var variable called score can be updated
- However, in the next example we can declare a var variable with the same name without causing an error
- This can cause bugs if you accidentally declare the same variable twice

Goodbye Var, Hello Let & Const

Let and const are **block** scoped

```es5
// ES5
var birthYear = 1990;
if (birthYear > 1900) {
    var age = 2018 - birthYear;
    console.log('You are ' + age + '!'); // You are 28!
}
console.log(age); // 28
```

```es6
// ES6
const birthYear = 1990;
if (birthYear > 1900) {
    let age = 2018 - birthYear;
    console.log(`You are ${age}!`); // You are 28!
}
console.log(age); // age is not defined
```

- Let and const are different from var because they are block scoped
- In the ES5 example, we have an if statement with a var variable called age
- If we console.log() age outside of that if statement, it will give us 28
- Since the if statement is not a function, the age variable leaks out into the global scope
- In the ES6 example, we have changed the var variable to a let variable
- If we try to console.log() age outside of the if statement, it will say age is not defined
- Since let is block scoped, it stays inside the if statement
- Blocks are basically anything inside curly brackets

Goodbye Var, Hello Let & Const

- Let variables can be **updated** but **not redefined** in the same scope
- Const variables **cannot be updated** or **redefined**
- Properties of an object declared with const *can* change

```
const firstName = 'Melinda';
let currentCity = 'Atlanta';

currentCity = 'Asheville';
console.log(currentCity); // Asheville

let currentCity = 'Asheville';
// Identifier 'currentCity' has already been declared

firstName = 'Jamie'; // Assignment to constant variable
const firstName = 'Melinda';
// Identifier 'firstName' has already been declared
```

```
const dog = {
    breed: 'Bull Terrier',
    age: 11
}

dog.age = 12;
console.log(dog.age); // 12
```

- **Let variables can be updated but not redefined in the same scope**
- Const is short for constant
- **Const variables cannot be updated or redefined**
- In the example we have a const variable called firstName and a let variable called currentCity
- The variable declared with let can be updated
- But if you tried to re-declare the let variable, the console would give an error saying it has already been declared
- The variable declared with const can NOT be updated
- It will give you an error saying assignment to a constant variable
- **However, properties of an object declared with const can change**

`Template Strings`

- Template strings are created by using back ticks
- The ${ } syntax is used to interpolate the variables
- JavaScript can be used inside the curly brackets

```
// ES5
var firstName = 'Sam';
var graduationYear = 2008;
var sentence = 'My name is ' + firstName + ' and I graduated
from college ' + (2018 - graduationYear) + ' years ago!';

console.log(sentence); // My name is Sam and I graduated from
college 10 years ago!
```

```
// ES6
const firstName = 'Sam';
const graduationYear = 2008;
const sentence = `My name is ${firstName} and I graduated from
college ${2018 - graduationYear} years ago!`;

console.log(sentence); // My name is Sam and I graduated from
college 10 years ago!
```

- Template strings are a new way of creating strings in JavaScript
- In the ES5 example, creating a string is complicated and hard to read
- **Template strings are created by using back ticks**
- **The ${ } syntax is used to interpolate the variables**
- You can add a variable, run a function, run any JavaScript, or add more template strings inside the curly brackets

`Template Strings`

```
var beer = {
    style: 'Berliner Weisse',
    origin: 'Germany',
    abv: 3.5,
    taste: 'Sour and acidic with no hop bitterness'
}

// ES5
var markup = ' \
    <div class="beer"> \
        <h1>' + beer.style + '</h1> \
        <p>Origin: ' + beer.origin + '</p> \
        <p>ABV: ' + beer.abv + '</p> \
        <p>Taste: ' + beer.taste + '</p> \
    </div> \
';
```

```
const beer = {
    style: 'Berliner Weisse',
    origin: 'Germany',
    abv: 3.5,
    taste: 'Sour and acidic with no hop bitterness'
}

// ES6
const markup = `
    <div class="beer">
        <h1>${beer.style}</h1>
        <p>Origin: ${beer.origin}</p>
        <p>ABV: ${beer.abv}</p>
        <p>Taste: ${beer.taste}</p>
    </div>
`;
```

- Another great use for template strings is creating HTML in JavaScript
- Previously, to create a multi-line string you would have to escape the new lines
- Now you can just use a template string
- The spaces ARE included in the string

Arrow Functions =>

- Arrow functions are always **anonymous** functions
- Delete the **function** keyword and add a fat arrow =>
- **No parameter**: pass empty parenthesis
- **One parameter**: parenthesis are optional
- **Multiple parameters:** use parenthesis
- You can remove the **return** keyword and **{ }** to create an **implicit return** on one line

```
const numbers = [1, 2, 3];

const plusOne = numbers.map(function(number) {
    return number + 1;
});

const plusOne2 = numbers.map((number) => {
    return number + 1;
});

const plusOne3 = numbers.map(number => {
    return number + 1;
});

const plusOne4 = numbers.map(number => number + 1);

console.log(plusOne); // [2, 3, 4]
```

- Arrow functions have a shorter syntax than function expressions
- **Arrow functions are always anonymous functions**
- An anonymous function is a function that is declared without a name
- In the example, we have an array of numbers
- We are going to perform a map function on that array where we add one to each item in the array
- The map method returns an array after doing something to every item in the original array
- To change the regular function to an arrow function:
- **Delete the function keyword and add a fat arrow =>**
- In this example, there is only one parameter so we can take out the parenthesis
- If there is no parameter, you must pass empty parenthesis
- If there are multiple parameters, you must use parenthesis
- If there is only a return value, you can use an implicit return
- **You can remove the return keyword and { } to create an implicit return on one line**
- These function expressions return the same array [2, 3, 4]
- Arrow functions are more concise

Arrow Functions =>

- An arrow function does not change the value of **this**
- Arrow functions inherit the value of **this** from the parent scope
- In the example, the parent scope is the window

```
const button = document.querySelector('.button');

button.addEventListener('click', function() {
    console.log(this); // <button
    class="button">Click me!</button>
});

button.addEventListener('click', () => {
    console.log(this); // Window
});
```

- **An arrow function does not change the value of this**
- **This** is whatever is to the left of addEventListener
- If you console.log() *this* inside of the addEventListener with the regular function, it will give you the button
- With the arrow function, the value of *this* is the window because it is inherited from the parent scope
- **Arrow functions inherit the value of this from the parent scope**
- **In the example, the parent scope is the window**

When Not to Use Arrow Functions

- **Click handlers**: by using a regular function, the value of **this** will be the element that was clicked on
- **Object methods**: by using a regular function, the value of **this** will be the object
- **Prototype methods**: a prototype method allows you to add a method to a class after it has been created. By using a regular function, the value of **this** will be the class
- **Arguments object**: you don't have access to the arguments object if you use an arrow function

- Don't start using arrow functions everywhere without understanding the pros and cons
- Arrow functions do not replace regular functions
- When not to use arrow functions:
- Click handlers
- Object methods
- Prototype methods
- The arguments object contains an array of the arguments passed to the function

Default Parameters

Default function parameters allow default values to be used if arguments are undefined when a function is called

```
// ES5
function calculateArea(width, length) {
    if (width === undefined) {
        width = 5;
    }
    if (length === undefined) {
        length = 10;
    }
    return width * length;
}

var area = calculateArea();
console.log(area); // 50
```

```
// ES6
function calculateArea(width = 5, length = 10) {
    return width * length;
}

const area = calculateArea();
console.log(area); // 50
```

- **Default function parameters allow default values to be used if arguments are undefined when a function is called**
- In the ES5 example, we have a function called calculateArea which has two parameters called width and length
- If you call the function without passing any arguments, the parameters will be undefined
- So you can create an if statement that checks if width is undefined and if it is true then width will be 5
- With ES6, we can easily set the default values inside the parameters
- Since no arguments are passed to the function, the arguments are undefined and the function will use the default parameters

For...of Loop

- The **for...of** statement creates a loop over iterable objects such as arrays, strings, maps, and sets
- On each iteration a value of a different property is assigned to the **variable**
- The **iterable** is the object we are iterating over

```
const buffet = ['🍕', '🍔', '🌭', '🍾'];

// ES5
for (var i = 0; i < buffet.length; i++) {
    console.log(buffet[i]); // 🍕 🍔 🌭 🍾
}

// ES6
for (const food of buffet) {
    console.log(food); // 🍕 🍔 🌭 🍾
}
```

- **The for…of statement creates a loop over iterable objects such as arrays, strings, maps, and sets**
- In our example we have an array of emojis called buffet
- If we want to console.log() every item in the array, we would use a for loop
- In the ES5 example, the for loop is hard to read and understand
- **With the for of loop, on each iteration a value of a different property is assigned to the variable**
- Variable = food
- **The iterable is the object we are iterating over**
- Iterable = buffet
- Both of these loops do the same thing but the for of loop is easier to understand
- Cannot iterate over an object
- Able to use break and continue

Array Methods

- The **Array.from()** method returns an array from any object with a length property or an iterable object
- The **Array.of()** method creates a new array with any number of arguments

```
const array1 = Array.from('dog');
console.log(array1); // ["d", "o", "g"]

const array2 = Array.of(1, 2, 3, 4);
console.log(array2); // [1, 2, 3, 4]
```

- There are new array methods in ES6
- Array.from() and Array.of() are for the Array object which is a global object that is used in the construction of arrays
- **The Array.from() method returns an array from any object with a length property or an iterable object**
- With array1, we are using Array.from() on a string called dog
- It will turn all of the letters from that string into an array
- **The Array.of() method creates a new array with any number of arguments**
- In the example array2, we use Array.of() and pass in four numbers, it will give us an array with all of those numbers

Array Methods

- The **find()** method returns the value of the first element in the array that passes a testing function
- The **findIndex()** method returns the index of the first element in the array that passes a testing function

```
const ages = [12, 16, 17, 25, 14];
const adult = ages.find(age => age > 18);
console.log(adult); // 25

const dogs = ['pug', 'corgie', 'beagle', 'lab'];
const index = dogs.findIndex(breed => breed === 'lab');
console.log(index); // 3
```

- **The find() method returns the value of the first element in the array that passes a testing function**
- In the first example, we have an array of ages
- Then we are going to use the find method to look for the first number that is greater than 18
- We are using an arrow function
- **The findIndex() method returns the index of the first element in the array that passes a testing function**
- In our example, we have an array of dogs
- We use the findIndex() method to find the first string that matches ‘lab’ and it will give us the index
- Both of these methods execute the function once for each element in the array
- If it finds an element where the function returns true, it returns the value or index of that element (and does not check the remaining values)
- These methods do not change the original array

Destructuring

Destructuring is a JavaScript expression that makes it possible to extract data from arrays, objects, maps, and sets into distinct variables

Destructuring

```es5
// ES5
var car = {
    make: 'Honda',
    model: 'Civic',
    year: 2012
}

var make = car.make;
console.log(make); // Honda

var model = car.model;
console.log(model); // Civic
```

```es6
// ES6
const car = {
    make: 'Honda',
    model: 'Civic',
    year: 2012
}

const { make, model } = car;

console.log(make); // Honda
console.log(model); // Civic
```

- We are taking the make and model properties from the car object and turning them into two new variables that are scoped to the parent block
- In the ES5 example, we have an object with three properties and we will declare a var variable and assign it to the property in the car object
- In the ES6 example, we have the same object but we are using the destructuring syntax to create our variables
- We are using the same names as the properties from the object to create the variables and then we are assigning that to the car object

Destructuring

```es5
// ES5
var contact = ['info@test.com', '123 Main Street', '123-456-7890'];

var email = contact[0];
var address = contact[1];
var phone = contact[2];

console.log(email, address, phone);
// info@test.com 123 Main Street 123-456-7890
```

```es6
// ES6
const contact = ['info@test.com', '123 Main Street', '123-456-7890'];

const [email, address, phone] = contact;

console.log(email, address, phone);
// info@test.com 123 Main Street 123-456-7890
```

- For arrays, you just swap out the curly brackets for square brackets

Destructuring

- You can access nested data
- The object property can be assigned to a variable with a different name
- You can provide default values if an object property is not defined

```
const rescue = {
    name: 'Furtastic Pet Rescue',
    dog: {
        name: 'Gunner',
        details: {
            breed: 'German Shepherd',
            gender: 'male'
        }
    }
}

const { breed: dogBreed, adoptable = true } = rescue.dog.details;

const { dog: { details: { breed: dogBreed, adoptable = true } } } = rescue;

console.log(dogBreed); // German Shepherd
console.log(adoptable); // true
```

- Here we have a complex object with nested data
- **You can access nested data with destructuring**
- I have shown two examples of how to access nested data
- These are the same but I think the first one is easier to understand
- **You can rename your variables if you don’t want to use the name of the property**
- We are renaming breed to dogBreed
- **You can provide default values if an object property is not defined**
- Since there is no adoptable property, it uses the default value of true

Destructuring

- The convertCups function returns an object
- The returned object is immediately destructured and new variables are created
- The order doesn't matter and variables can be left out if they are not needed

```
function convertCups(cups) {
    const conversions = {
        pints: cups / 2,
        quarts: cups / 4,
        gallons: cups / 16
    }
    return conversions;
}

const { pints, quarts, gallons } = convertCups(5);

console.log(pints); // 2.5
console.log(quarts); // 1.25
console.log(gallons); // 0.3125
```

- Here is an example of using destructuring with a function
- This function takes cups and converts them into pints, quarts, and gallons
- **The convertCups function returns an object**
- **The returned object is immediately destructured and new variables are created**
- **The order doesn't matter and variables can be left out if they are not needed**

Spread and Rest...

Spread syntax takes an iterable and expands it into its elements

```
const string = [...'hello'];
console.log(string); // ["h", "e", "l", "l", "o"]

const shoppingList1 = ['milk', 'eggs', 'bread'];
const shoppingList2 = ['bananas', 'yogurt', 'cereal'];

const combinedList1 = shoppingList1.concat(shoppingList2);
console.log(combinedList1); // ["milk", "eggs", "bread", "bananas", "yogurt", "cereal"]

const combinedList2 = [...shoppingList1, ...shoppingList2];
console.log(combinedList2); // ["milk", "eggs", "bread", "bananas", "yogurt", "cereal"]

const newShoppingList = [...shoppingList1, 'zucchini', ...shoppingList2];
console.log(newShoppingList); // ["milk", "eggs", "bread", "zucchini", "bananas", "yogurt", "cereal"]
```

- **Spread syntax takes an iterable and expands it into its elements**
- The spread syntax is just those three dots
- In the first example, spread syntax takes a string and spreads it into individual letters
- In the next example, we are going to combine two arrays
- We can use the concat() method to combine the arrays
- Or we can put the two arrays inside of square brackets while using the spread syntax and they are combined into one array
- An advantage to using spread syntax is that other items can easily be added to the new array

Spread and Rest...

Spread syntax can be used to change a NodeList to an array

```
const services1 = Array.from(document.querySelectorAll('.services ul li'));
console.log(services1); // [li, li, li]

const services2 = [...document.querySelectorAll('.services ul li')];
console.log(services2); // [li, li, li]
```

- **Spread syntax can be used to change a NodeList to an array**
- querySelectorAll returns all elements in the document that matches the CSS selectors as a NodeList object
- A NodeList is array-like but it doesn’t have the same methods as an array
- We can use the Array.from() method to create a new array
- Or we can use spread syntax to create a new array

Spread and Rest...

The rest parameter collects multiple elements and condenses them into a single element

```
const northCarolinaCities = ['Raleigh', 'Asheville', 'Charlotte', 'Wilmington'];

const [capital, ...cities] = northCarolinaCities;
console.log(capital, cities); // Raleigh ["Asheville", "Charlotte", "Wilmington"]

function calculateTotal(tax = 0.07, ...items) {
    console.log(items); // [9.99, 14.99, 5.99]
    const itemsWithTax = items.map(item => Math.round((item * tax + item) * 100) / 100);
    return itemsWithTax.reduce((total, amount) => total + amount);
}

const total = calculateTotal(undefined, 9.99, 14.99, 5.99);
console.log(total); // 33.14
```

- The rest parameter is the opposite of spread syntax
- **The rest parameter collects multiple elements and condenses them into a single element**
- In our first example we have an array called northCarolinaCities
- Using destructuring, we created a variable for the capital and the rest are saved into an array called cities
- In our next example, we have a function called calculateTotal which will take an unknown number of items and create a total cost
- We have created a default parameter for the tax which is 0.07
- If we pass undefined to the function, it will use the default value
- Since we don’t know how many items there will be, we can use the rest parameter to create an array with all the items
- Then we can use map to add the tax to each item
- Then we can use reduce to add all of the items together and return the total
- The rest parameter is useful when you don’t know how many parameters there will be

Classes

- Classes provide a simpler syntax to create objects and leverage prototype-based inheritance
- Classes can be declared with a class **expression** or a class **declaration**
- The **constructor** method is a special method for creating and initializing an object created with a class

```
// Class declaration
class Person {
    constructor(name, job) {
        this.name = name;
        this.job = job;
    }
}

// Class expression
const Person = class {
    constructor(name, job) {
        this.name = name;
        this.job = job;
    }
};
```

- **Classes provide a simpler syntax to create objects and leverage prototype-based inheritance**
- **Classes can be declared with a class expression or a class declaration**
- To declare a class, you use the class keyword with the name of the class
- Class expressions can be named or unnamed
- In the example, I am showing the two ways of declaring a class
- **The constructor method is a special method for creating and initializing an object created with a class**
- There can only be one special method with the name "constructor" in a class

Classes

- An instance of a class is created by using the **new** keyword and then the constructor function is called
- **Methods** are attached to an instance of a class and can be added directly in the class

```
class Animal {
    constructor(name, sound) {
        this.name = name;
        this.sound = sound;
    }
    // Method
    makeSound() {
        console.log(this.sound);
    }
}

const cow = new Animal('cow', 'moo');
cow.makeSound(); // moo
```

- In this example, we are creating a class called Animal which will create two properties called name and sound
- **An instance of a class is created by using the new keyword and then the constructor function is called**
- So we are creating a new instance of the class called cow and passing in cow and moo as the arguments
- **Methods are attached to an instance of a class and can be added directly in the class**
- In the example, the makeSound() method just console.log()'s the sound
- You do not need commas between properties or methods in a class

Classes

- With getter and setter methods, logic can be included when retrieving or setting the value of a property
- A **setter** method receives a value and can perform logic on that value, then it either updates an existing property or stores it to a new property
- When a **getter** method is called, a property value is computed and returned, but this value is not updated or stored anywhere

```
class Animal {
    constructor(name, sound) {
        this.name = name;
        this.sound = sound;
    }
    // Getter
    get species() {
        return this._species;
    }
    // Setter
    set species(species) {
        this._species = species;
    }
}

const cow = new Animal('cow', 'moo');
cow.species = 'Bos taurus';
console.log(cow.species); // Bos taurus
```

- **With getter and setter methods, logic can be included when retrieving or setting the value of a property**
- **A setter method receives a value and can perform logic on that value, then it either updates an existing property or stores it to a new property**
- To use the setter method, just type the name of the object followed by a dot and the name of the setter method and assign it to a value
- Setters always receive one parameter which is the value of the property to be set
- In the example, we are using the species() setter method and passing in the string ‘Bos Taurus’
- The name of the property cannot be the same as a getter or setter method, so we have to create a ‘backing property’ which is the name of the setter function but with an underscore
- If you try to console.log(cow.species) before creating the getter method, you would get undefined
- The getter method returns the value of the backing property
- **When a getter method is called, a property value is computed and returned, but this value is not updated or stored anywhere**
- The value of the property computed in the getter method can be accessed like a regular property using dot or bracket notation
- In the example, we can console.log() cow.species and it will give us ‘Bos Taurus’

Sets

- The Set object lets you store *unique* values of any type
- The **add** method adds items to the set
- The **has** method checks if the set contains a certain element
- The **delete** method removes an element from a set
- The **clear** method removes all items from the set

```
const people = new Set();
people.add('Chris');
people.add('James');
people.add('Sam');

console.log(people); // {"Chris", "James", "Sam"}

people.has('Chris'); // true

people.delete('Sam');
console.log(people); // {"Chris", "James"}

people.clear(people);
console.log(people); // {}
```

- ES6 has introduced new data structures
- **The Set object lets you store unique values of any type**
- A value in the Set may only occur once
- It’s different from an array because you can’t access the items individually and it’s not index-based
- You can think of a set as a list of items we can add to, remove from, or loop over
- **The add method adds items to the set**
- **The has method checks if the set contains a certain element**
- **The delete method removes an element from a set**
- **The clear method removes all items from the set**

Maps

- The Map object holds key-value pairs
- The **set** method adds or updates an element with a specified key and value to the map
- The **has** method checks if the map contains a key
- The **get** method returns an element from a map
- The **delete** method removes a key-value pair from the map

```
const dogs = new Map();
dogs.set('Cypress', 12);
dogs.set('Champ', 3);
dogs.set('Grover', 5);
console.log(dogs);
// {"Cypress" => 12, "Champ" => 3, "Grover" => 5}

dogs.has('Champ'); // true

dogs.get('Grover'); // 5

dogs.delete('Champ');
console.log(dogs);
// {"Cypress" => 12, "Grover" => 5}
```

- **The Map object holds key-value pairs**
- A map is similar to a set except it has a key and a value instead of just values
- The keys and the values can be anything
- One benefit to using a map over an object is that you can use a for of loop
- **The set method adds or updates an element with a specified key and value to the map**
- **The has method checks if the map contains a key**
- **The get method returns an element from a map**
- **The delete method removes a key-value pair from the map**

Resources

ES6 for Everyone es6.io

Learn ES2015 babeljs.io

Introduction to ES6 learn.co

ES6 in Depth ponyfoo.com

freeCodeCamp freecodecamp.org