COVALENCE

Advanced JavaScript

More on Variables

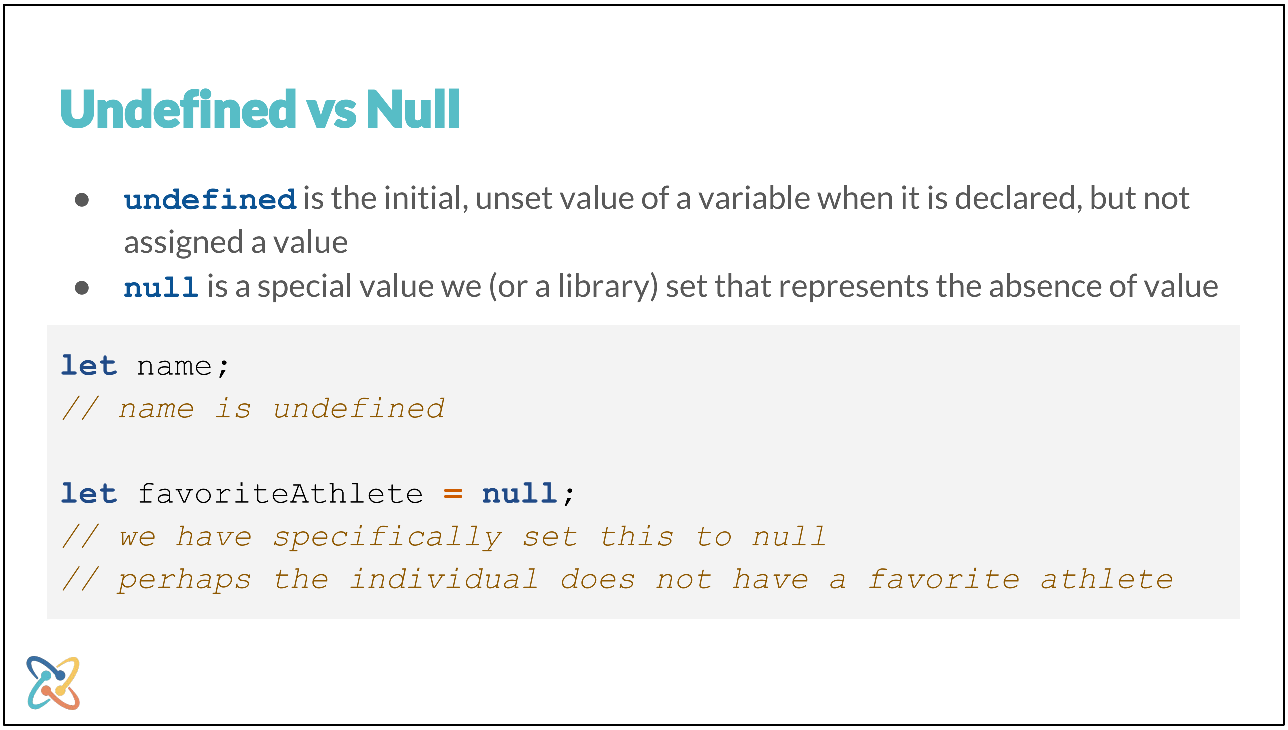
Variable Declarations and Assignments

- So far, we've usually declared our variables and assigned values to them at the same time
- But they can be done separately. For example, here is a variable declaration:

```
let name;
```

- When a variable is declared, but not assigned a value, it is **`undefined`**
- And here is an assignment statement for that variable later:

```
name = 'John';
```



Undefined vs Null

- `undefined` is the initial, unset value of a variable when it is declared, but not assigned a value
- `null` is a special value we (or a library) set that represents the absence of value

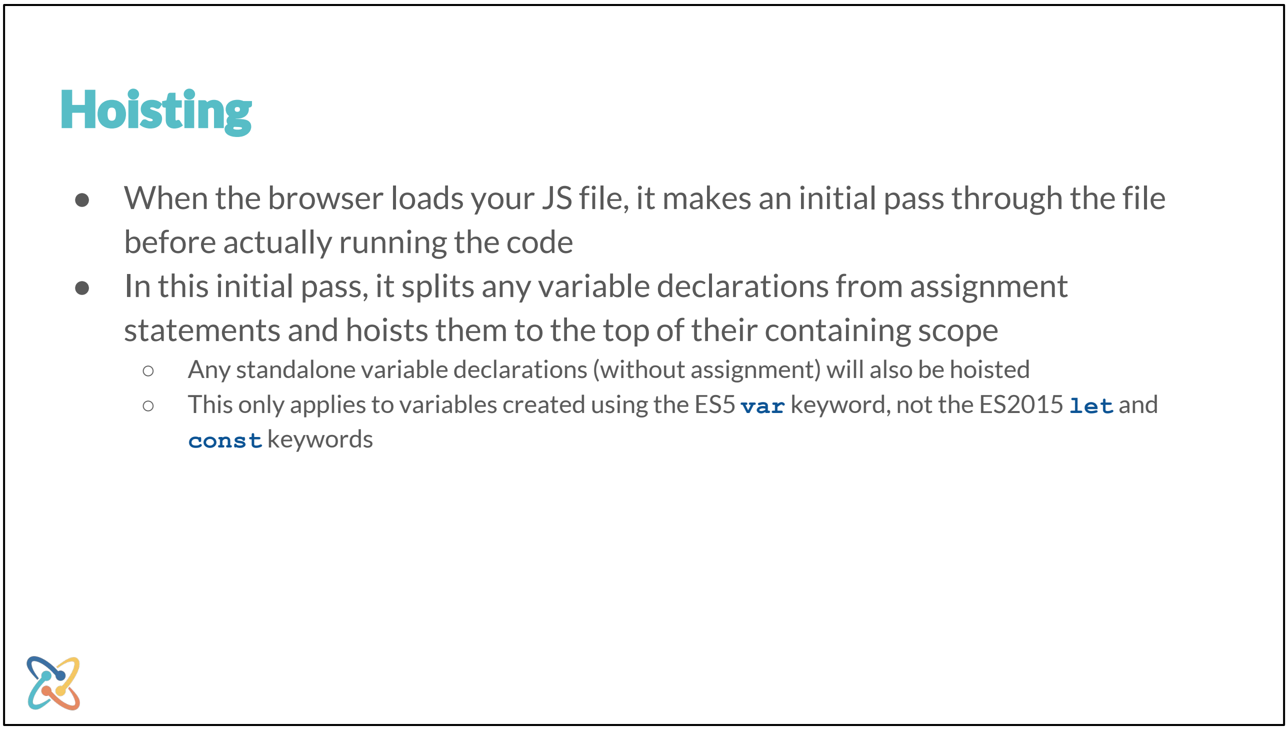
```
let name;
// name is undefined

let favoriteAthlete = null;
// we have specifically set this to null
// perhaps the individual does not have a favorite athlete
```

Think of undefined as meaning we just haven't set something yet, and null as meaning we are specifically saying there is no value

COVALENCE

Advanced JavaScript

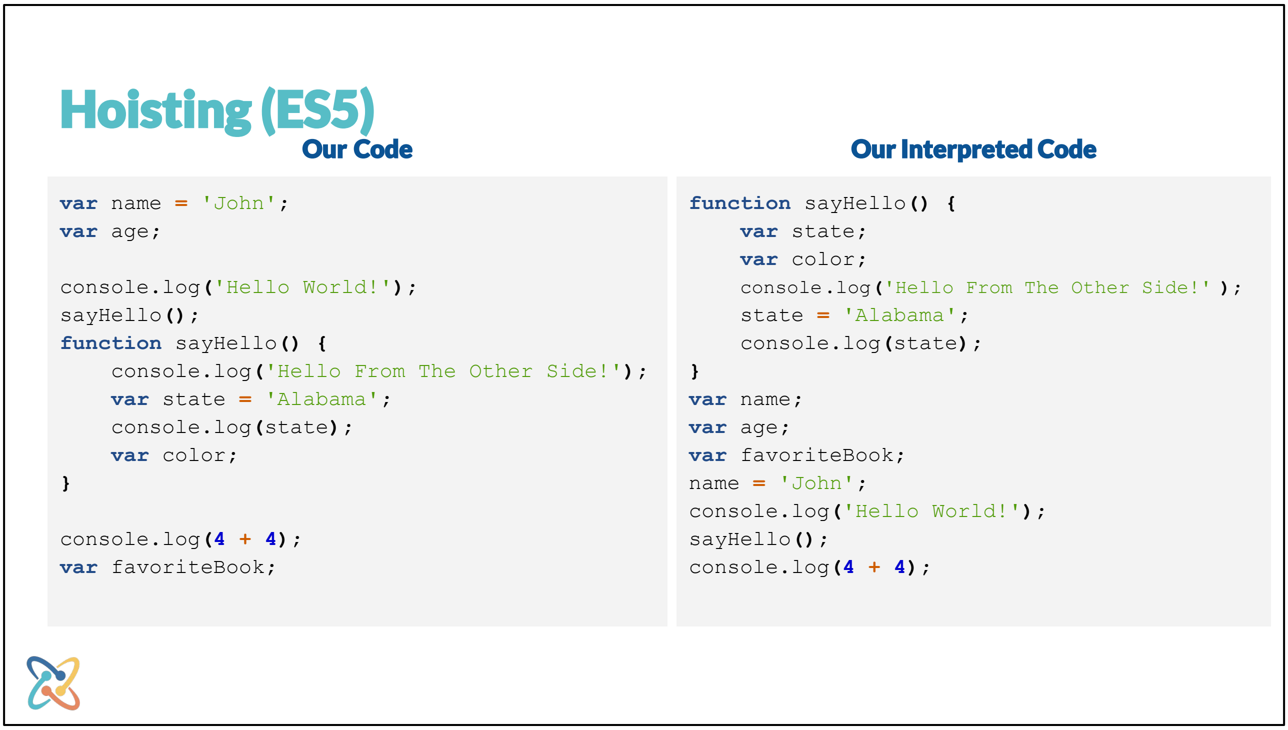
Hoisting



Hoisting

- When the browser loads your JS file, it makes an initial pass through the file before actually running the code
- In this initial pass, it splits any variable declarations from assignment statements and hoists them to the top of their containing scope
  - Any standalone variable declarations (without assignment) will also be hoisted
  - This only applies to variables created using the ES5 `var` keyword, not the ES2015 `let` and `const` keywords

- http://adripofjavascript.com/blog/drips/variable-and-function-hoisting.html
- Advanced, thorough explanation: https://scotch.io/tutorials/understanding-hoisting-in-javascript



Hoisting (ES5)

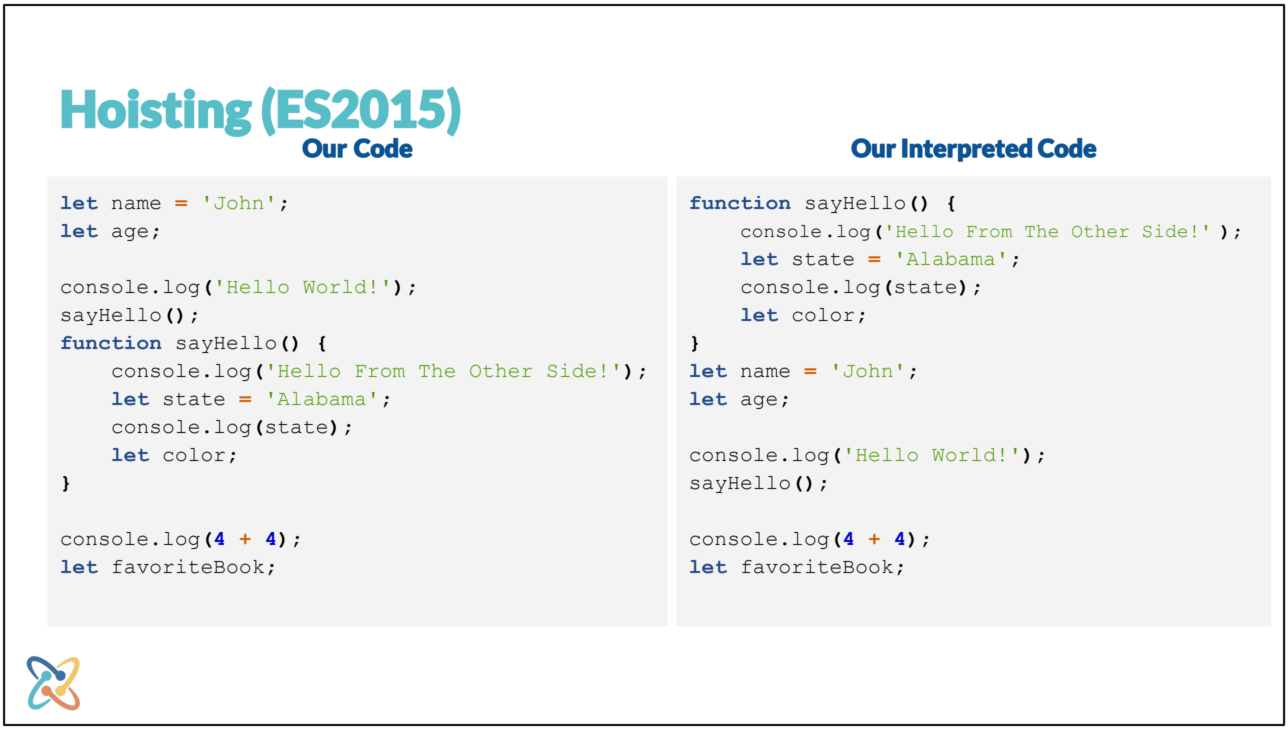
Our Code

```
var name = 'John';
var age;

console.log('Hello World!');
sayHello();
function sayHello() {
    console.log('Hello From The Other Side!');
    var state = 'Alabama';
    console.log(state);
    var color;
}

console.log(4 + 4);
var favoriteBook;
```

Our Interpreted Code

```
function sayHello() {
    var state;
    var color;
    console.log('Hello From The Other Side!' );
    state = 'Alabama';
    console.log(state);
}
var name;
var age;
var favoriteBook;
name = 'John';
console.log('Hello World!');
sayHello();
console.log(4 + 4);
```

- Every function and variable is hoisted to the top of each scope (functions end up on top of variable declarations)
- (the same hoisting process occurs within each function too)
- Then the assignments and other statements execute as intended
- Remember that in ES5 JavaScript:
    - *Variable declarations* (formed using var) are hoisted to the top of the scope in which the variable was created (global, function scope, etc.)
    - Function declarations are also hoisted to the top of the scope in which the function was declared (global, another function, etc) and placed ahead of hoisted variable declarations



Hoisting (ES2015)

Our Code

```
let name = 'John';
let age;

console.log('Hello World!');
sayHello();
function sayHello() {
    console.log('Hello From The Other Side!');
    let state = 'Alabama';
    console.log(state);
    let color;
}

console.log(4 + 4);
let favoriteBook;
```

Our Interpreted Code

```
function sayHello() {
    console.log('Hello From The Other Side!' );
    let state = 'Alabama';
    console.log(state);
    let color;
}
let name = 'John';
let age;

console.log('Hello World!');
sayHello();

console.log(4 + 4);
let favoriteBook;
```

- Remember that in ES2015 JavaScript:
    - *Variable declarations* formed using let or const are NOT hoisted
    - *function declarations* are still hoisted, however

COVALENCE

Advanced JavaScript

More on Functions

Function Review

- Recall that we use functions to group one or more statements together
- We can call the function at a later time, causing all the statements within it to execute
- A function that doesn't have a name is called an **anonymous function**
- Usually event listeners are set up with anonymous functions

```
btn.addEventListener('click', function() {
    alert('Clicked!');
});
```

Function Names

- If a function is not anonymous, it must have a name
- Something that you may not have realized is that a function name is a variable
- There are two ways of creating a function
    - Function declarations start with the function keyword, and are hoisted
    - Function expressions are anonymous functions assigned into variables
        - When using var, the variable declaration itself will be hoisted, but not the function body

```
// Will be hoisted
function sayHello() {
    console.log('Hello World!');
};

// Will not be hoisted at all (let)
let sayHello = function() {
    console.log('Hello World!');
};

// variable declaration sayHello (var) will be
hoisted, but not the function body
var sayHello = function() {
    console.log('Hello World!');
};
```

Calling A Function

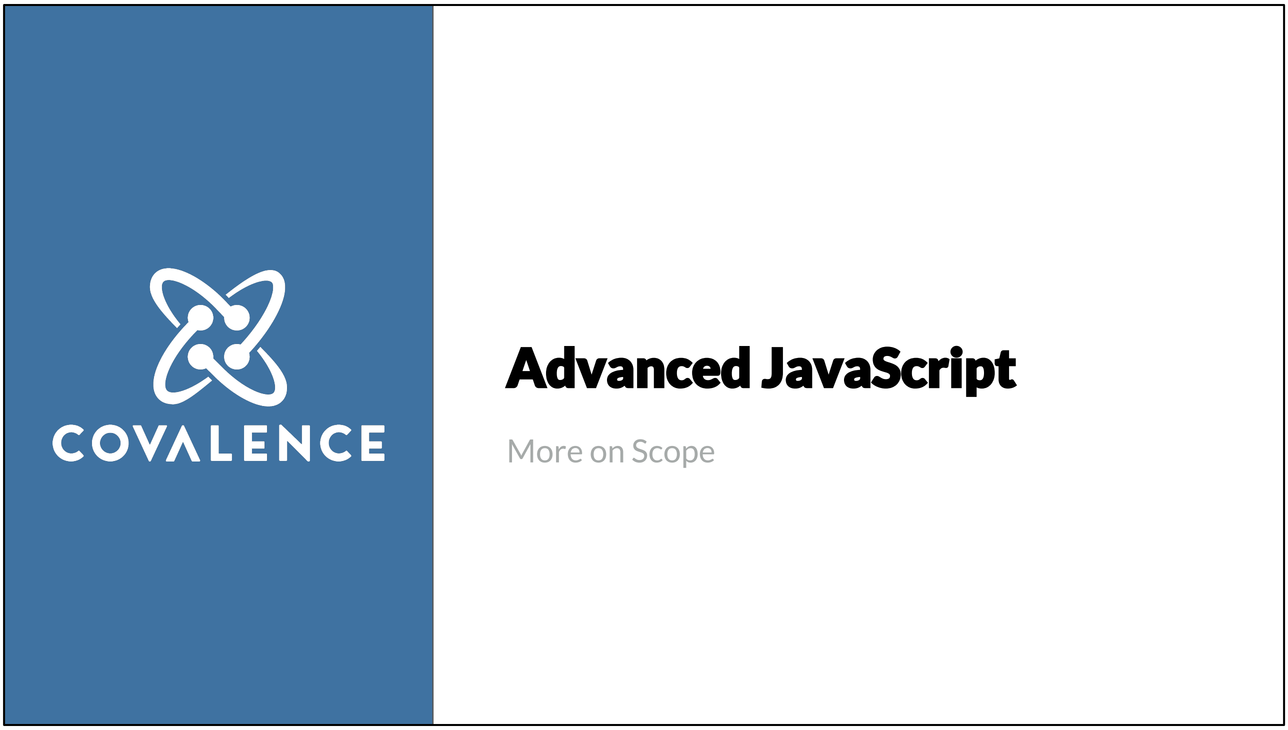
- To call a named function (make it execute), you type the variable name holding the function, followed by the call function operator: `()`
- No matter whether the function created using a function declaration or a function expression, we call the function the same way:

```
sayHello()

variableName()
```

Function Returns

- You have the option of having a function return a value
- When a function returns, it stops executing and *returns in place*
- We jump back to where the function was called
- If a value is returned, the value "takes the place" of the function call
- Functions can return numbers, strings, booleans, objects, even functions!

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

let sum = add(5,2);

// think of it as becoming:
let sum = 7;
```



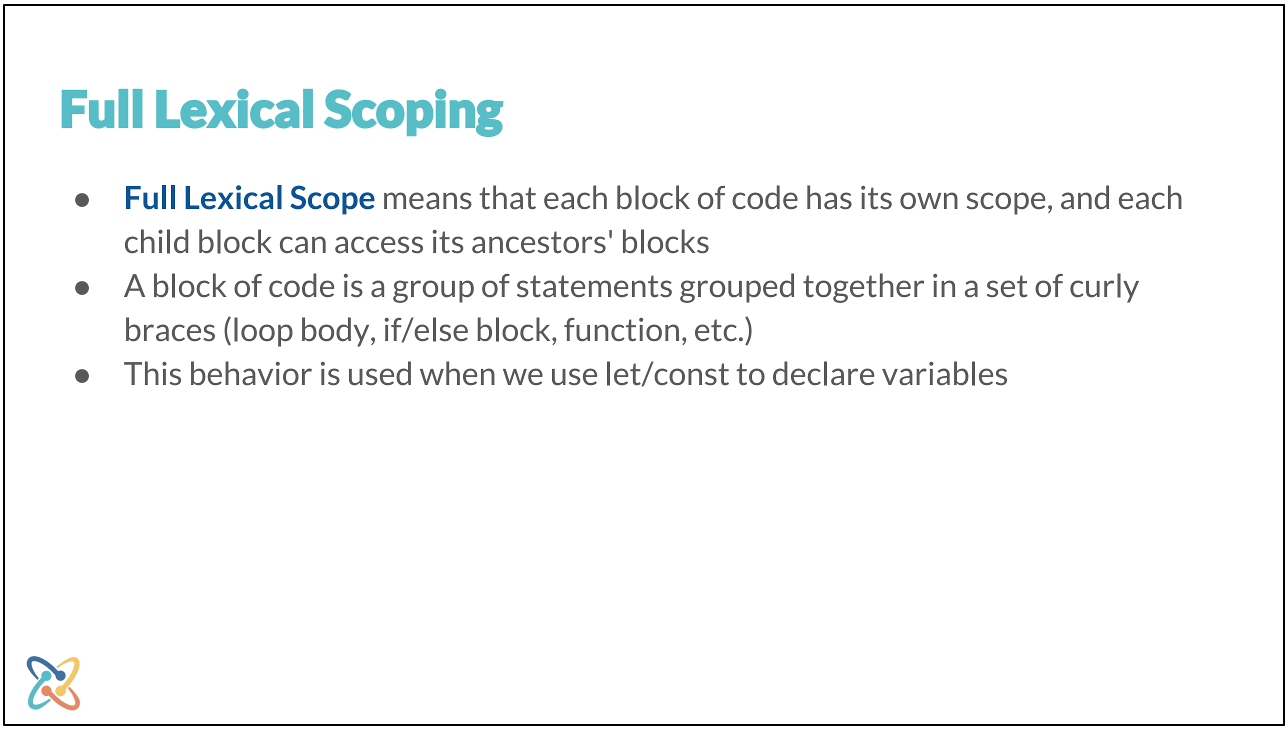
COVALENCE

Advanced JavaScript

More on Scope

Scope

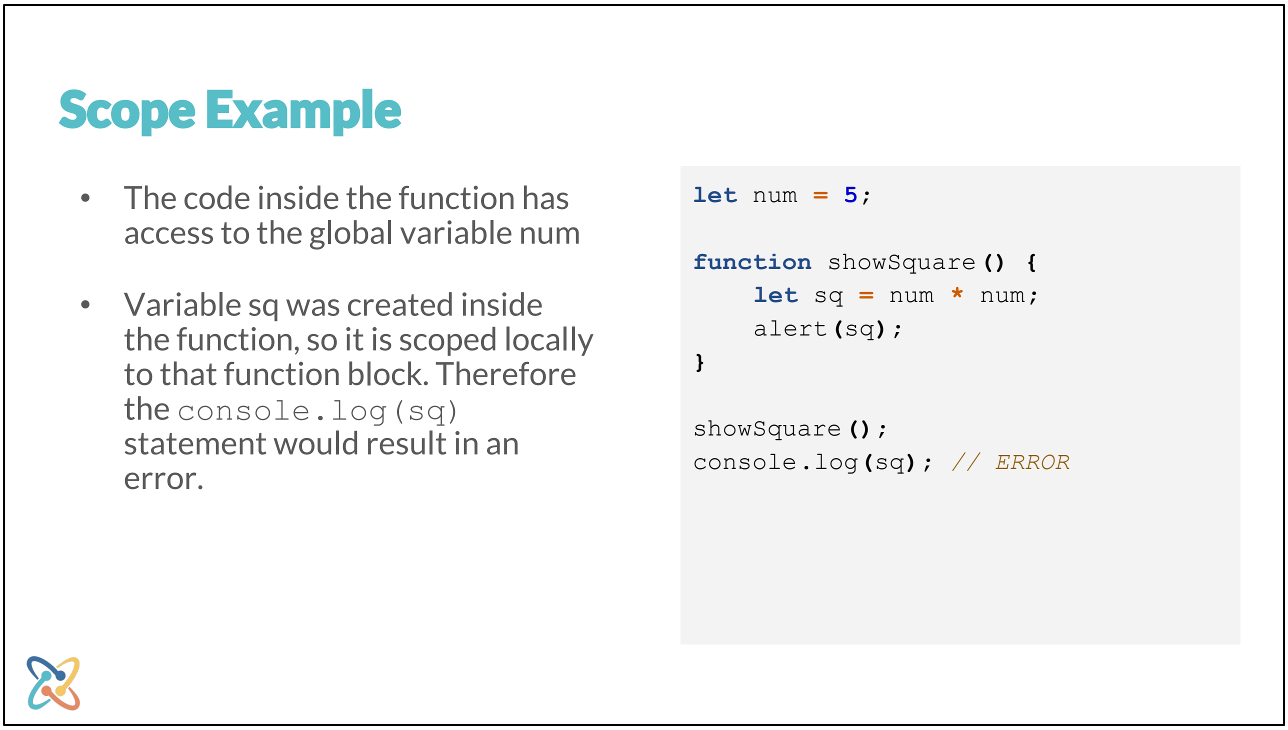
- Scope is the collection of variables and functions accessible to you in a particular point in the code
- Recall that JavaScript uses different scoping behavior depending on if you use var (ES5) or let/const (ES2015)
- We will be reviewing the ES2015 scoping mechanism



Full Lexical Scoping

- **Full Lexical Scope** means that each block of code has its own scope, and each child block can access its ancestors' blocks
- A block of code is a group of statements grouped together in a set of curly braces (loop body, if/else block, function, etc.)
- This behavior is used when we use let/const to declare variables

- **Blocks get their own scope** (If/Else, loop bodies, etc.)
- Any variable/function not declared in a block, is in the global scope
- Code in a block can access its local scope and the global scope
- Code not inside a block can only access the global scope



Scope Example

- The code inside the function has access to the global variable num
- Variable sq was created inside the function, so it is scoped locally to that function block. Therefore the `console.log(sq)` statement would result in an error.

```
let num = 5;

function showSquare() {
    let sq = num * num;
    alert(sq);
}

showSquare();
console.log(sq); // ERROR
```

- The variable (sq) is created inside the showSquare function, and is destroyed once the showSquare function finishes executing
- The variable (sq) cannot be accessed outside of the showSquare function

Gotcha: Automatic Globals

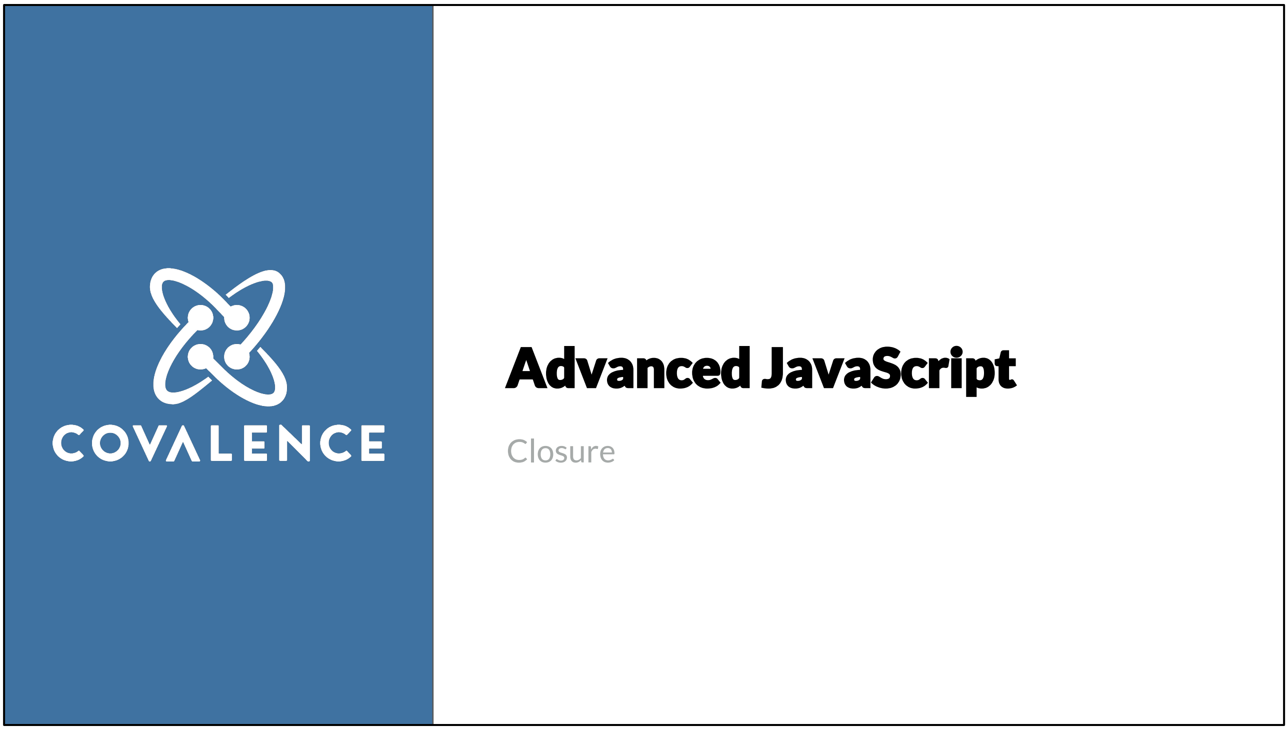
- Remember that when creating a variable for the first time, always use the let/const keywords
- Leaving it off can have unintended side effects

Gotcha: Automatic Globals

- Creating a variable inside a function without using let/const will cause the variable to go onto the global scope, not the local scope. When that happens, the variable is called an **automatic global**

- That can cause a headache for you and especially for unsuspecting developers who may be maintaining your code later
- There is **never** a valid reason for not using the let/const keyword when creating a variable. JUST USE IT.

```
let num = 5;

function showSquare() {
    sq = num * num;
    alert(sq);
}

showSquare();
console.log(sq); //25 (auto global)
```



Advanced JavaScript

Closure

Closure

- A firm understanding of scope and functions is required in order to understand closures
- A **closure** is the combination of a function and the lexical environment (scope) within which that function was declared
- A closure is not something you see; it is a behavioral concept of functions in JavaScript
- In basic terms, when a function is declared in JavaScript, that function retains a snapshot of all variables and functions it can access (i.e. its scope)
  - In other words, a function "remembers" all the functions and variables it had access to when it was created
- This snapshot is what we refer to when we use the term closure

Closure: Example

```
function makeFunc() {
  let name = 'Covalence';
  function displayName() {
    alert(name);
  }
  return displayName;
}

let myFunc = makeFunc();
myFunc();
```

- Recall that functions can return any type of data, *including other functions*
- In this example, `makeFunc` is a function that returns a function
- Since functions return in place, the function returned by `makeFunc` gets stored in the variable `myFunc`
- When you have the name/variable that holds a function, you simply type its name followed by the function call operator: `()`
- This will display an alert on the page that says Covalence

Closure: Example

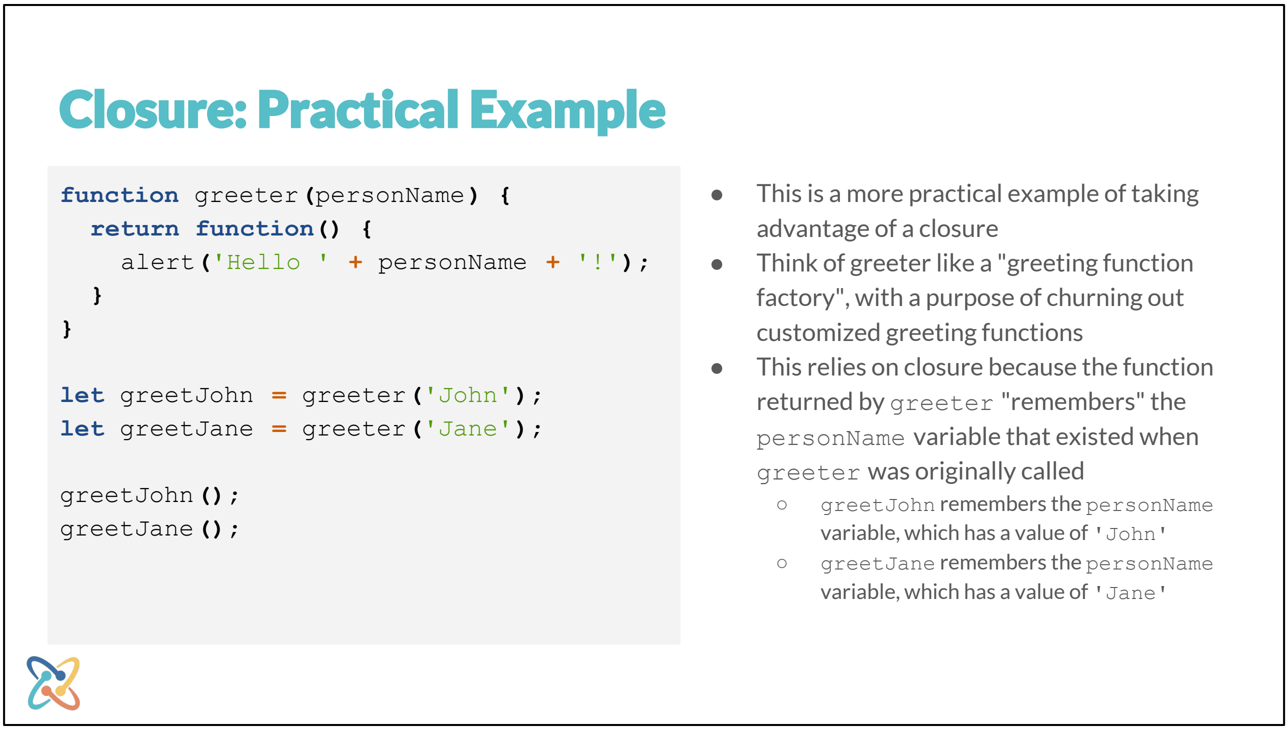
```
function makeFunc() {
  let name = 'Covalence';
  function displayName() {
    alert(name);
  }
  return displayName;
}

let myFunc = makeFunc();
myFunc();
```

- If you have previous programming experience, your instincts may be screaming at you at this point
- In many other languages, variables created within a block are discarded when that block stops executing
  - So variable `name` would have been discarded before `myFunc()` could have been called
- This is mostly true in JavaScript, but we have that sneaky concept of closure to thank for this code's ability to work

Closure: Example

```
function makeFunc() {
  let name = 'Covalence';
  function displayName() {
    alert(name);
  }
  return displayName;
}

let myFunc = makeFunc();
myFunc();
```

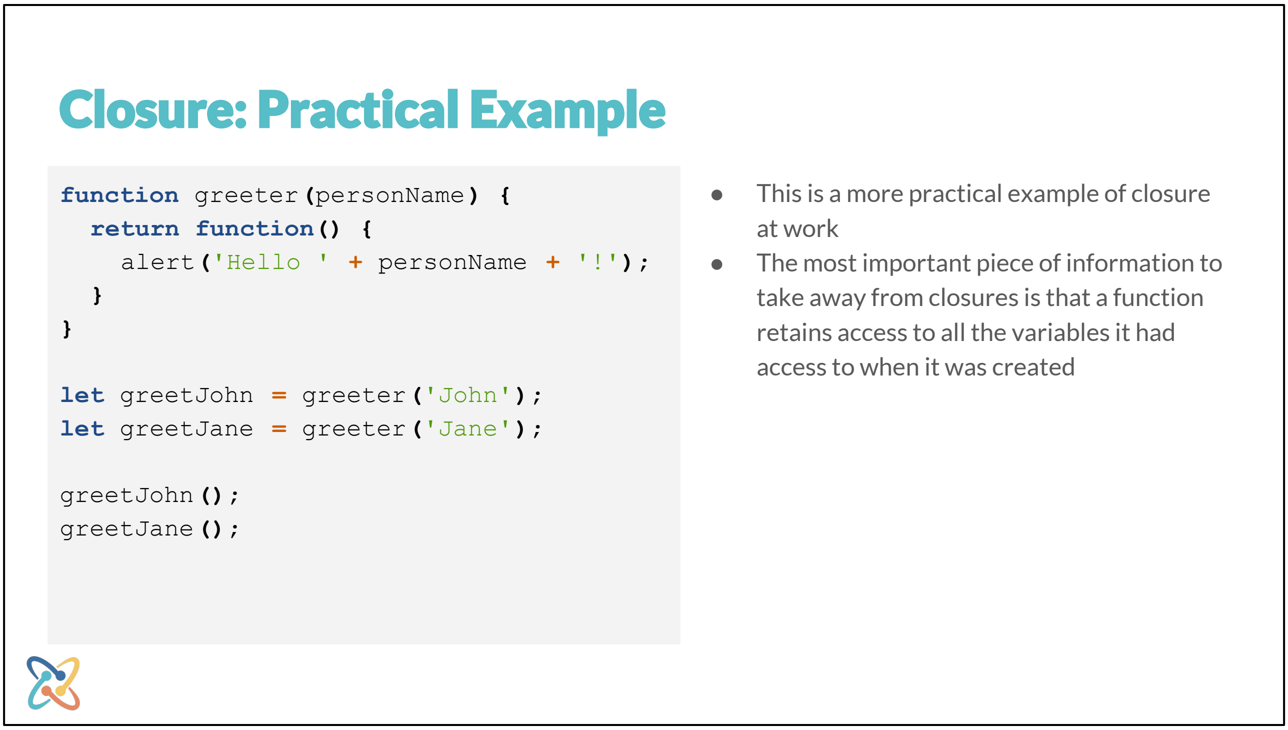
- When `makeFunc()` was called, it started executing
- First, the variable name was declared and assigned a value of `'Covalence'`
- Next, the function `displayName` was declared
- At that moment in time, a closure was created
- The `displayName` function now retains access to the `name` variable because that variable was within the lexical scope the function could "see" when it was created



Closure: Practical Example

```
function greeter(personName) {
  return function() {
    alert('Hello ' + personName + '!');
  }
}

let greetJohn = greeter('John');
let greetJane = greeter('Jane');

greetJohn();
greetJane();
```

- This is a more practical example of taking advantage of a closure
- Think of greeter like a "greeting function factory", with a purpose of churning out customized greeting functions
- This relies on closure because the function returned by `greeter` "remembers" the `personName` variable that existed when `greeter` was originally called
  - `greetJohn` remembers the `personName` variable, which has a value of `'John'`
  - `greetJane` remembers the `personName` variable, which has a value of `'Jane'`



Closure: Practical Example

```
function greeter(personName) {
  return function() {
    alert('Hello ' + personName + '!');
  }
}

let greetJohn = greeter('John');
let greetJane = greeter('Jane');

greetJohn();
greetJane();
```

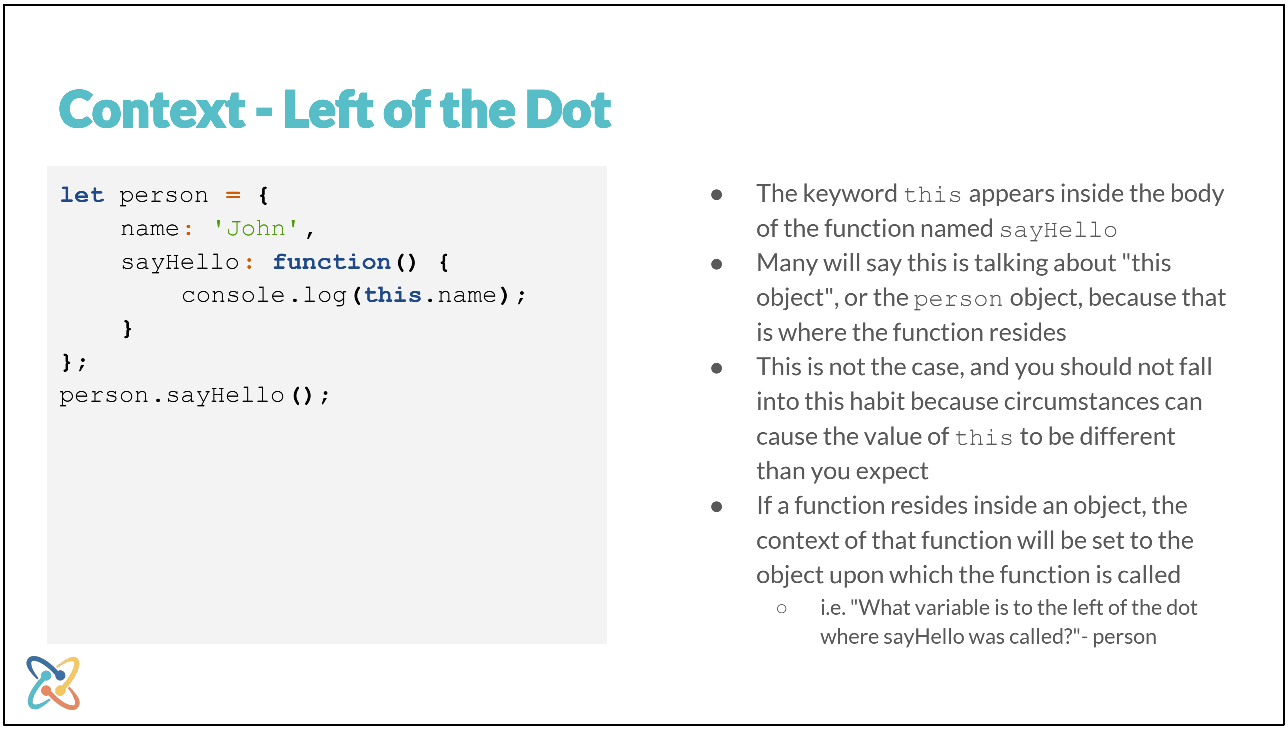
- This is a more practical example of closure at work
- The most important piece of information to take away from closures is that a function retains access to all the variables it had access to when it was created

Advanced JavaScript

Function Context: What is it and what is `this`?

Function Context

- **Function Context** has nothing to do with *where* a function resides in our code, even though it frequently seems that way
- Context is determined based on *how* a function is called
- The special keyword **`this`** refers to the function context
- The function context is determined based on several different guidelines



Context - Left of the Dot

```
let person = {
    name: 'John',
    sayHello: function() {
        console.log(this.name);
    }
};
person.sayHello();
```

- The keyword `this` appears inside the body of the function named `sayHello`
- Many will say this is talking about "this object", or the `person` object, because that is where the function resides
- This is not the case, and you should not fall into this habit because circumstances can cause the value of `this` to be different than you expect
- If a function resides inside an object, the context of that function will be set to the object upon which the function is called
    - i.e. "What variable is to the left of the dot where sayHello was called?"- person

- So when the function runs, we console.log(this.name)
- this will be set to the person object (left of .sayHello()), so the name property of the person object will be accessed
- In this case, that value is 'John'

Context - DOM Instigator

```
let btn =
document.getElementById('big-button');

btn.addEventListener('click', function() {
    console.log(this);
});
```

- Consider this example
- We didn't actually call the function that runs the console.log statement
- In this case, a DOM event listener is running the specified anonymous function on our behalf, whenever the button is clicked
- When a function is called by a DOM event listener, the function's context is set to the DOM element for which the event occurred (e.g. the button that was clicked, the div that was moused over, etc.)

Context - Otherwise

```
function testThis() {
    console.log(this);
}
testThis();
```

- In this example, the function was not called on an object or from an event listener
- In this case, the function context will be the global object, which is the `window` object on web browsers

Context is King

```
function testThis() {
    console.log(this);
}

testThis();

let btn =
document.getElementById('big-button');

btn.addEventListener('click', testThis);
```

- Consider this example
- When `testThis` runs from our function call, `testThis()`, we will see the global object logged to the console
- However, if you click the button, that same function will run with different results
    - This is because context varies depending on ***how*** a function is called
    - When we click the button, a DOM event listener is running the function on our behalf, and the context is set to the DOM element causing the action (in this case, the clicked button)

COVALENCE

Advanced JavaScript

Manipulating function context through call, apply, and bind

Manipulating Function Context

- In addition to the guidelines for determining function context (the value of `this`), you also need to know how to manually set a function's context to the value of your choosing
- This can be accomplished through 3 related functions: `call`, `apply`, and `bind`

call

- `call` is a function built into every single function in JavaScript, even functions you create
- Consider if variable `d` were a string and we wanted to call the built-in `toUpperCase()` function, we would use: `d.toUpperCase()`
- Similarly, if we had a function named `sayHello` and we wanted to use `.call` on it, we would use: `sayHello.call(...)`

call

```
function sayHello(name) {
    console.log(name);
    console.log(this);
}
sayHello('John');
```

- But what does it do?
- `call` allows us to call (run) a function, optionally specifying the function context and any arguments we wish to pass in a comma separated list
- This example (NOT using `call`) would log John, and then log the window object, because that is the function context based on the guidelines

call

```
function sayHello(name, age) {
    console.log(name);
    console.log(age);
    console.log(this);
}
sayHello.call('Jane', 'John', 27);
```

- The first argument to `.call` is always what we want to set the function's context to when it runs
- In this example, we are setting the function context to the string `'Jane'`
- After that, any additional arguments can be specified separated by commas and will be dropped into the function's parameters
- In this example, we will see `'John'`, then `27`, then `'Jane'` log to the console

call With No Arguments

```
function doSomething() {
    console.log(this);
};
doSomething.call('John');
```

- You do not need to send arguments; you can simply set the function context
- In this example, the string `'John'` would log to the console

apply

```
function sayHello(name, age) {
    console.log(name);
    console.log(age);
    console.log(this);
}
sayHello.apply('Jane', ['John', 27]);

// OR
let args = ['John', 27];

sayHello.apply('Jane', args);
```

- `apply` is the sister function to call
- Operates the exact same way, but function arguments are passed as an array instead of a comma separated list
- `this` will be `'Jane'`, `name` will be `'John'`, and `age` will be `27`

bind

- If `call` and `apply` are sisters, **`bind`** is their cousin
- `call` and `apply` immediately call the function, manipulating `this` and passing in the specified arguments
- `bind` creates and returns a copy of the function, with `this` manipulated (binded/bound) and arguments already set, but the function does not yet execute

bind

```
function sayHello(name, age) {
    console.log(name);
    console.log(age);
    console.log(this);
}

let greeter = sayHello.bind('Jane');
greeter('John', 27);
```

- Note that `bind` does not cause the function to run
- Instead, a copy of the function is made, with the context (`this`) already set to the specified value
- Now when the function is called, we will see `'John'`, and then `27`, and then `'Jane'` log to the console