JavaScript - Understanding the Weird Parts

Contents

Execution Contexts and Lexical Environments

Conceptual Aside: Syntax Parsers, Execution Contexts, and Lexical Environments

Syntax parser

- A program that reads your code and determines what it does and if its grammar or syntax is valid.
- You have your code that you've written and there's a program that is going to convert what you've written into a real set of computer instructions, something the hardware can physically understand.
- There's a compiler or an interpreter between those two things (your code and computer instructions) and part of that is a syntax parser.

Lexical environment

- Where something sits physically in the code you write.
- The word ‘lexical’ means ‘having to do with words or grammar’.
- Lexical environment exists in programming languages in which ‘where’ you write something is important.

Execution context

- A wrapper to help manage the code that is running.
- There are lots of lexical environments, areas of the code that you are looking at physically. But which one is currently actually running, is managed via what's called execution contexts.
- And an execution context contains your code, the running code. It's running your code, but it also can contain things beyond what you've written in your code because remember, your code is being translated, being processed by a whole other feature, a whole other set of programs that someone else wrote. And so, it's executing your code and it can do other things as well.

Conceptual Aside: Name/Value Pairs and Objects

- A **Name/Value** pair is a name which maps to a unique value. And the value can contain other name/value pairs (nested name/value pairs).
- An **object** is a collection of Name/Value pairs. That's the simplest possible definition of an object when you're talking about JavaScript.

The Global Environment and The Global Object

- Whenever code is run in JavaScript, its run inside an execution context.
- Meaning a wrapper that the JavaScript engine, the program that other people wrote, that's parsing and looking at and verifying and executing your code. That wraps the code that you've written, it wraps the currently executing code in an execution context.
- The base execution context is your **global execution context** and it has a couple of special things that come along for the ride.
- When we say global, we're talking about the thing that's accessible everywhere to everything in you're code, it's global.
- So the global execution context creates two things for you. It creates a **Global Object**. And it creates a special variable called **'this'**.
- The JavaScript engine is creating these two things for you whenever your code is run, because your code is wrapped inside an execution context.
- There is always a **global object** when you're running JavaScript. In the case of browsers, it's the **window** object. Each browser tab has its own global execution context.
- In case of browser the global execution context creates global object called window and 'this' variable. And at the global level those two things are equal so window object = this
- **When we say global, in JavaScript that means not inside a function.**
- In JavaScript, when you create variables and functions, and if you're not inside a function, those variables and functions get attached to the global object.
- Refer – 01-global-environment

The Execution Context - Creation and Hoisting

- In JavaScript, the variables and functions are to some degree available even though they're written later in the code.
- The reason JavaScript behaves this way is the execution context is created in two phases.
- The first phase is **creation phase**. In this phase, it creates global execution context (global object and this variable) and also creates and outer environment.
- Also it sets up the memory space for the variables and functions used in your code and this is called as "**Hoisting**".
- All this means is that before your code begins to be executed line by line, the JavaScript engine has already set aside memory space for the variables that you've created in that entire code that you've built, and all of the functions that you've created as well.
- So those functions and those variables exist in memory. So when the code begins to execute line by line, it can access them.
- So the JavaScript engine when it sets up the memory space for a variable, it doesn't know what its value will ultimately end up being until it starts executing its code. So

instead, it puts a placeholder called **undefined**. That placeholder means “oh, I don't know what this value is yet”.

- **All variables in JavaScript are initially set to undefined, and functions are sitting in memory in their entirety.**
- So when the code begins executing, those things are actually already sitting in memory because it looked at your code and already preset things up to be ready for the code to start executing.
- Refer – 02-executation-context-hoisting

Conceptual Aside: JavaScript and 'undefined'

- 'undefined' is actually a special value that JavaScript has within it internally that means that the variable hasn't been set.
- **Never set yourself a variable equal to undefined. It is meant to be set by JavaScript in creation phase.**
- It's better to let ‘undefined’ mean that I, the programmer, never set this value. That will really help you when debugging code.
- The error "Uncaught ReferenceError: b is not defined" simply means the variable is not present in the memory space which was created in the creation phase.
- Refer – 03-javascript-and-undefined

The Execution Context - Code Execution

- And the second phase, is the **execution phase**.
- In the execution phase we already have all those things set up that we had before. And now it runs your code line by line, interpreting it, converting it, compiling it, executing it on the computer into something the computer can understand.
- Refer – 04-executation-context-execution-phase

Conceptual Aside: Single Threaded, Synchronous Execution

- Single Threaded – That means that one command is being executed at a time.
- Synchronous means, in the purposes of programming, one at a time. So one line of code being executed for synchronous execution at a time and for our purposes, in order that it appears.
- **JavaScript is single threaded, synchronous execution in its behavior.**

Function Invocation and the Execution Stack

- Invocation – That just means running a function or calling a function.
- Anytime you execute or invoke a function in JavaScript, a new execution context is created and put on the execution stack. Once the function execution is one, the execution context is popped off the stack. And whichever one is on top is the one that's currently running.

Functions, Context, and Variable Environments

- **Variable environment** is where the variables live that you've created and how they relate to each other in memory.
- Every execution context has its own variable environment.
- Refer – 05-variable-environment

The Scope Chain

- **Variable environment** is where the variables live that you've created and how they relate to each other in memory.
- Every execution context has its own variable environment.
- However when we request a variable or when we do something with the variable, JavaScript does more than just look in the variable environment of the currently executing context.
- Every execution context has a reference to its **outer (lexical) environment**.
- JavaScript does something special, it cares about the lexical environment when it comes to the outer (lexical) environment reference that every execution context gets. And when you ask for a variable while running a line of code inside any particular execution context, and if it can't find that variable, it will look at the outer (lexical) environment reference and go look for variables there somewhere down below it in the execution stack (**scope chain**). And that outer environment reference where that points is going to depend on where the function sits **lexically**.
- That whole chain is called the Scope Chain. Scope means, where I can access a variable. And the chain is those links of outer (lexical) environment references.
- Refer – 06-scope-chain

Scope, ES6, and let

- Execution context, execution environment, variable environment, lexical environment, all these things ultimately define "scope".
- Scope is where a variable is available in your code.
- ES6 introduced new keyword called "let". "**let**" allows the JavaScript engine to use **block scoping**.

- So you can declare a variable with “let”, and during the execution phase where it's created, the variable is still placed into memory and set to undefined. However, you're not allowed to use “let” variable until the line of code is run during the execution phase that actually declares the variable. If you try to use it before the declaration statement, you will get an error.

Asynchronous Callbacks

- Asynchronous simply means more than one at a time.
- So we may be dealing with code that's executing and that starts off some other code to execute, and that may start other code executing and all of those pieces of code are actually executing within the engine at the same time.
- So since **JavaScript is synchronous, how is this handling those asynchronous events**?
- While the rendering engine, JavaScript engine and HTTP requests are running asynchronously inside the browser, what's happening inside just the JavaScript engine is synchronous.
- When any event happens in browser e.g. click event, http request completion, etc., then those events are put into **event queue**. This event queue is looked up by JavaScript.
- When the execution stack is **empty**, then JavaScript periodically looks at the event queue. It waits for something to be there, and if something is there, it looks to see if a particular function (event callback function) should be run when that event was triggered. So it creates the execution context for that (callback) function when that event happens. And so that event is processed and the next event in the queue moves up, and so on and so forth. But t**he event queue won't be processed until the execution stack is empty**, that is until JavaScript is finished running all of that other/main code line by line.
- So Javascript isn't really asynchronous. What happens is the browser asynchronously puts things into the event queue, but the code that is running inside JavaScript Engine is still running line by line. And then when the execution stack is empty, then it processes the events. It waits for them and sees an event. And if an event causes a function to be created and executed, then it will appear on the execution stack and run like normal.
- The JavaScript engine won't look at the event queue until the stack is empty. So that means long-running functions can actually interrupt events being handled. This is how JavaScript **synchronously** is dealing with the fact that **asynchronous** events are happening.
- Any events that happen outside of the JavaScript engine get placed into the event queue, and if the execution stack is empty, if JavaScript isn't working on anything else currently, it'll process those events in the order they happened.
- Refer – 07-asynchronous-callbacks

Types and Operators

Conceptual Aside: Types and Javascript

- **Dynamic Typing** – That means that you don't tell the JavaScript engine what type of data a variable holds. Instead, the engine figures it out while your code is running, while it's executing.
- So a single variable can hold different types of values at different times during the execution of your code, because it's all figured out during execution.
- Java is statically typed language.
- JavaScript is dynamically typed language.
- This turns out to be quite powerful, and can also cause you some problems if you don't understand how JavaScript is going to make decisions as a result of dynamic typing.

Primitive Types

- **Primitive type** is a type of data that represents a single value. In other words, something that isn't an object.
- There are 6 primitive types in JavaScript –
    - **undefined** – Undefined represents a lack of existence. And it's what the JavaScript engine initially sets variables to and it will stay undefined unless you set the variable to have a value. You should not use it. It is meant to be used by JavaScript.
    - **null** – also represents lack of existence. However you can use it when you want to say that something doesn't exist, that the variable has no value.
    - **boolean** – true or false
    - **number** – floating point number.
    - **string** – single or double quotes can be used.
    - **symbol** – New in ES6.

Conceptual Aside: Operators

- An **operator** is actually a **special function** that is syntactically, or that is written differently than regular functions are written in your code.
- Generally, operators take two parameters and return one result.
- JavaScript operators (which are in the end just functions) use **infix notation**. Infix means that the function name, that is the operator, sits between the two parameters.

Operator Precedence and Associativity

- Operator precedence just means which operator function gets called first where there's more than one on the same line of executable code.
- And functions then are called in an order of precedence. The higher precedence wins.
- Associativity means what order an operator gets called in (left to right or right to left) when functions/operators have the same precedence.
- Refer – PDF Operator Precedence In Javascript.pdf

Conceptual Aside: Coercion

- **Coercion** means converting a value from one type to another.
- This happens quite a bit in JavaScript, because it's dynamically typed.
- E.g.
  var a = 1 + 2; // => 3
  var b = 1 + '2'; // => 12. Here number 1 is coerced to string '1'.

Comparison Operators

- Refer – PDF. Equalty Comparison And Sameness.pdf
- Refer Equality comparisons and sameness - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness
- **Default coercions** –
  ```
  Number('1');        // 1
  Number(true);       // 1
  Number(false);      // 0
  Number(undefined);  // NaN
  Number(null);       // 0
  ```
- Wouldn't it be nice if we had a way to not coerce under a certain, very common circumstance? Solution is Strict equality (===).
- **Some weird output with double Equality (==)** –
  ```
  1 == 1;  // true
  1 == '1'; // true
  1 == true; // true
  null == 0; // false
  null < 1; // true
  Number(null); // 0
  "" == 0; // true
  "" == false; // true
  ```
- == can make your code very difficult to anticipate what's going to happen. The solution is using strict equality (===) or strict inequality (!==)

- Strict equality is a standard operator, but doesn't try to coerce the values. If the two values are not the **same type**, they're not equal and returns false.
- E.g.
  3 === 3; // true
  3 === "3"; // false
- So using the triple equals will prevent us from having some odd potential errors in our code when comparing values.
- **Use triple equals when making equality comparisons. Don't use double equals.**
- Refer - 08-comparison-operators

Existence and Booleans

- We can use coercion to our own advantage and check to see if a variable has a value.
- E.g. Boolean(undefined);   // false
  Boolean(null);   // false
  Boolean("");   // false
  Boolean(0);   // false
  Boolean(1);   // true
  Boolean("abcd");   // true
- Refer – 09-existence-and-booleans

Default Values

- The OR operator's special behavior, in that if you pass to it two values that can be coerced to true and false, it will return the first one that coerces to true. E.g. 0 || 1; => 1
- Some examples –
  undefined || "Hello";   // "Hello"
  null || "Hello";   // "Hello"
  false || "Hello";   // "Hello"
  0 || "Hello";   // "Hello"
  true || "Hello";   // true
  1 || "Hello";   // 1
- So **we can use one line OR expression to set a default value, if not passed.**
- Refer – 10-default-values

Objects and Functions

Objects and the Dot

- An object is just collection of key value pairs.
- Objects can have properties and methods.
- With compute access operator '[' ']', you can dynamically decide what property you're trying to get.
- Always recommended to use the dot operator unless you really need to access a property using some kind of dynamic string.
- Refer – 11-objects-and-dot

Objects and Object Literals

- Object literal is shorthand way to quickly create objects.
- Object literal syntax turns out to be really powerful. It can make for some very clean looking and easy to write code.
- Refer – 12-object-literals

Framework Aside: Faking Namespaces

- In modern coding, a namespace is a container for variables and functions.
  It's just a holder, a container.
- And typically it's used to keep variables and functions with the same name separate.
- JavaScript doesn't have namespaces. However because of the nature of objects, we don't need namespaces as a feature. We can fake it.
- Faking namespaces is common across frameworks and libraries.
- Refer – 13-faking-namespaces

JSON and Object Literals

- JSON is inspired by object literal syntax in JavaScript and so it's called JavaScript Object Notation.
- But JSON and object literal are NOT the same.
- In previous years, **data was sent over the internet** in various formats and the format that was landed upon for a while was **xml**.
- The **problem with XML** is when you're dealing with download times, how fast something is and how much data, how much bandwidth you are using, XML has a lot of extra unnecessary characters (opening and closing tags, etc.) that make the amount of data that you're sending larger. That's a huge amount of wasted download bandwidth if you were dealing with a lot of data.
- The **solution** to this problem came in the form of JSON. Whatever in the xml can be easily formatted in the form of Javascript object literal format. And since we have to send

the object literal format over the internet, we needed to serialize it. Hence **JSON**!. The Javascript object is just stringified, meaning property names are wrapped in quotes to make it as a string and the value is anyway string or number.

- Because of this JSON has become defacto standard for data transfer between clients (e.g. browser) and server (e.g. node.js, REST).
- JavaScript does come with some **built in** functionality to transfer between the JSON and object literal.
  JSON.stringify() – converts the object to JSON string.
  JSON.parse() – converts the (JSON) string to JS object.
- Refer – 14-json-and-object-literal

Functions are Objects

- In JavaScript, functions are objects.
- **First class functions** – Everything you can do with other types, objects, strings, numbers, booleans, you can do with functions.
- First class functions change the way you can program. They can open up the horizons to completely different approaches to solving problems.
- It's a special type of object, because it has all the features of a normal object and has some other special properties. Just like any object in JavaScript, it resides in memory.
- You can attach properties and methods to a function. Because it's just an object. So I could attach a primitive, a name value pair, another object, and other functions.
- In JavaScript, the function object has some hidden special properties. Two important ones are its **name and code**.
- Now a function in JavaScript doesn't have to have a **name**. If it doesn't have name, it's called anonymous. But it can have a name. The other property **code** is where the actual lines of code that you've written sit.
- **So it isn't like the code that you write is the function. The function is an object with other properties. And the code that you write is just one of those properties** that you're adding onto it. What's special about that property is it's invocable (using ( ) braces) which means – "run this code".
- In browser console, type "greet.", you will see all available variables and methods for this function.
- Refer – 15-functions-are-objects

Function Statements and Function Expressions

- An **expression** is a unit of code that results in a value.
- A function expression or any expression in JavaScript ends up creating a value and that value doesn't necessarily have to save inside a variable.
- E.g.

var a = 3;
1+2;
Both these lines are expressions because both lines return some value.

- This concept of first class functions, where you can pass functions around, give functions to other functions, use them like you do variables, introduces an entirely new class of programming called **functional programming**.
- Refer – 16-function-expressions

Conceptual Aside: By Value vs By Reference

- **By value** – passing as arg, or referencing, or setting equal one value to another, by copying the value into a separate spot in memory. All **Premitives** in Javascript are passed by value.
- **By Reference** – passing as arg, or referencing, or setting equal one value to another, by referencing to existing spot in memory. All **Objects (including functions)** in Javascript are passed by reference.
- IMP: equals operator sets up new memory space (new address) if doesn't already exist in memory.
- Refer – 17-by-value-vs-by-reference

Objects, Functions, and 'this'

- "this" will be pointing at a different object or a different thing, depending on how the function is invoked.
- IMP Refer – 18-objects-functions-this

Case 1:

- Whenever we create a function (function expression or a function statement) at global level, then "this" (used inside that function) will point to the global object (window object in case of browser).

Case 2:

- In the case where a function is actually a method attached to an object, "this" keyword (used inside that method) becomes that object that the method is sitting inside of.

Case 3:

- However if there is another function inside method of an object. Then "this" keyword (used inside that function of the method of an object) points to global object (window object in case of browser).
- Many people think 'this' (used inside that function of the method of an object) should point to that object but 'this' here points to global object. So people think it as a bug in JS, but this is how JS works!

- To tackle this problem, one of the common patterns people use is using another variable (e.g. ‘self’ or ‘that’) to point to ‘this’ (that object). People use either ‘self’ or ‘that’ as a variable name in such cases.
- This pattern you’ll see very often if you’re working in any real-world JavaScript scenarios. However, the ‘let’ keyword, which will be an alternative to the var keyword, is meant to clear some of these problems up.
- No programming language is perfect. They all have their quirks, and JavaScript certainly isn’t an exception. But there are patterns we can use to get around any problems the programming language might have.

Conceptual Aside: Arrays – Collections of Anything

- In JavaScript, an Array can hold elements of different types (primitives, objects, function expressions).
- Refer – 19-arrays

'arguments' and spread

- When you execute a function, JavaScript engine sets up for you automatically a special keyword called “arguments”.
- “**arguments**” contains a list of all the values, of all the parameters that you pass to a function.
- **Arguments** is just another name for the parameters you pass to a function.
- “**arguments**” variable is array-like but not exactly JavaScript Array.
- Refer – 20-arguments-and-spread

Framework Aside: Function Overloading

- Function Overloading means a function of the same name that has different numbers of parameters.
- Now this doesn't really work in JavaScript because functions are objects. So that functionality isn't available in Javascript. But we have few patterns by which we can simulate such behavior.
- Refer – 21-function-overloading

Conceptual Aside: Syntax Parsers

- Remember that the code you write isn't directly run on the computer, but there's that intermediate program between your code and the computer that translates your code into something the computer can understand.
- The JavaScript engine on your browser for example does this. And it has different aspects and elements to it, one of them being a syntax parser, which reads your code and determines if it's valid, and what it is you're trying to do.

Dangerous Aside: Automatic Semicolon Insertion

- A Dangerous Aside is where we're touching a subject that's so dangerous that it's so easy to make a mistake and it's so hard to track down. And that you need to always avoid it.
- The syntax parser in JavaScript does something that tries to be helpful.
- Semicolons are optional in core JavaScript.
- So anywhere where the syntax parser expects that a semicolon would be, it will put one for you. (e.g. return statement)
- **You should always put your own semicolons.** Because you don't want the JavaScript engine to make that decision for you. You want to be certain that you are writing the code as it should be.
- Especially in the case of return, automatic semicolon insertion can cause a big problem in your code.
- Everywhere where you would expect to have a semicolon, you should put one to avoid this problem as much as possible.
- Refer – 22-automatic-semicolon-insertion

Framework Aside: Whitespace

- JavaScript is very liberal in accepting whitespace.
- As much as you can, comment up your code, make your code readable. Make your code understandable. You won't regret it in the future. ☺
- Refer – 23-whitespace

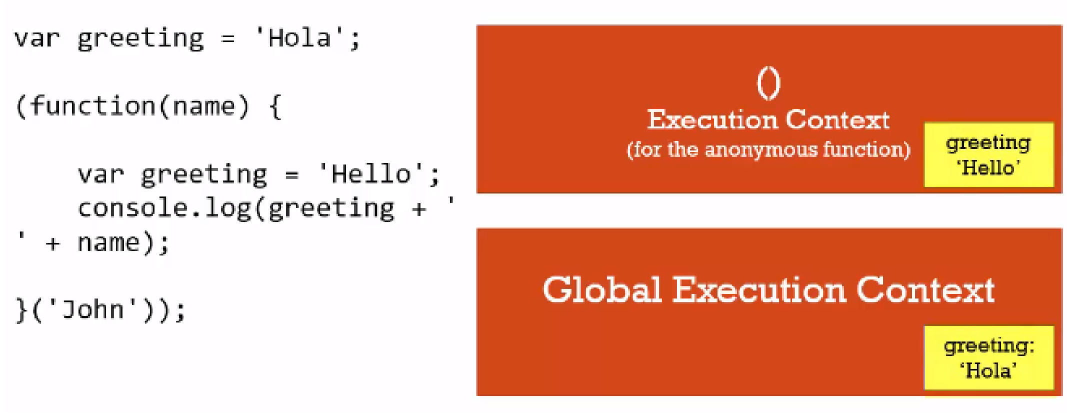
Immediately Invoked Functions Expressions (IIFEs)

- This invokes the function immediately after creating it.
- This is similar to writing below standalone, without storing the value to a variable.
  3;
  "test string";
- Generally we only use parenthesis '(' and ')' with expressions e.g. (3+4)*2. You never put a statement inside parentheses, like 'if ()'. It always is an expression, something that returns a value.
- So since the JavaScript engine knows that anything inside a parentheses must be an expression, it assumes that this the function wrapped inside parenthesis that you've written is a function expression.
- So using parentheses is to trick syntax parser from throwing errors.
- This turns out to be a wonderful tool in your arsenal, as a JavaScript developer.
- And you'll see this form, this style in almost every major framework and library that's out there today.

- IMP Ref – 24-immediately-invoked-function-expressions

Framework Aside: IIFEs and Safe Code

- IIFEs ensure that the code inside it is safe, in the sense that it does not interfere with, crash into, or be interfered by any other code that might be included in my application.
- This is because separate execution stack is created when an IIFE is invoked.



```
var greeting = 'Hola';

(function(name) {

    var greeting = 'Hello';
    console.log(greeting + '
' + name);

}('John'));
```

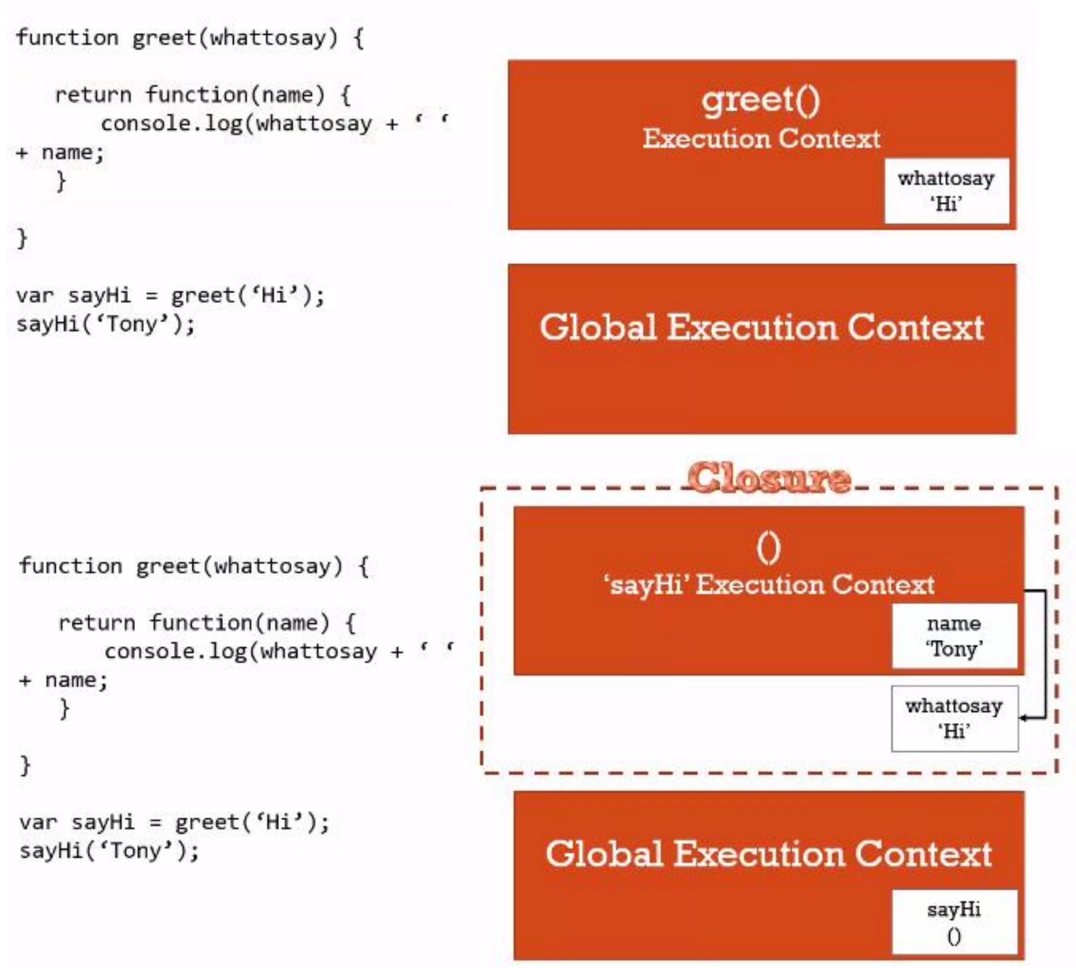
- And so, in a many libraries and functions, if you open their source code, the very first thing you'll see at the top is an opening parentheses and a function and at the very bottom, the closing parenthesis. Because it wraps all its code in an immediately invoked function.
- It's something that we can imitate in your own code to make sure we aren't colliding with other code when we're creating something reusable, by ensuring that we're not accidentally putting something into the global object.
- And if you specifically want access to the global object inside your IIFE, just pass that global object as an argument to your IIFE and then you can mess around with that global object since objects are passed by reference. So if you want something in your IIFE to be to be available everywhere, you can stick it on the global object.
- Refer – 25-iifes-and-safecode

Understanding Closures

- Every execution context has a space in memory, where the variables and functions created inside of it live. But what happens to that memory space when the execution context goes away (popped off the stack)? Under normal circumstances, the JavaScript engine would **eventually** clear it out with a process called **garbage collection**.
- But at the moment that execution context finishes, that memory space is still there somewhere in memory.
- Example –

```
function greet(whattosay) {

   return function(name) {
       console.log(whattosay + ' ' + name);
   }
}

var sayHi = greet('Hi');
sayHi('Sameer');
```

- Now when the inner function (or sayHi() function) expects variable ‘whattosay’, JS simply goes down the scope chain into the outer lexical environment reference. And even though the execution context of greet function is gone, the sayHi execution context still has a reference to the variables, to the memory space of its outer lexical environment.



```
function greet(whattosay) {

   return function(name) {
       console.log(whattosay + ‘ ‘
+ name;
   }

}

var sayHi = greet(‘Hi’);
sayHi(‘Tony’);
```

```
function greet(whattosay) {

   return function(name) {
       console.log(whattosay + ‘ ‘
+ name;
   }

}

var sayHi = greet(‘Hi’);
sayHi(‘Tony’);
```

- **In other words, even though the greet function ended, any functions created inside of it, when they are called, will still have a reference to that greet function's execution context memory space**.

- The phenomenon of “closing in” all the variables that a function is supposed to have access to, is called a **closure**. Those variables may be from outer function (different execution context). So even if the outer function execution context is popped off the execution stack, JavaScript engine still makes sure that inner function can still go down the scope chain to find any variables even though it's not even on the execution stack anymore.
- Closures are simply a feature of the JavaScript programming language. They just happen. It doesn't matter when we invoke a function. We don't have to worry if its outer environments are still running. **The JavaScript engine will always make sure that whatever function I'm running, that it will have access to the variables that it's supposed to have access to**, that its scope is intact.
- Refer – Understanding Closure – [26-closures]
- Refer – Understanding Closure – a complex scenario - [27-closures-2]

Difference between var and let

- Refer – https://stackoverflow.com/questions/762011/whats-the-difference-between-using-let-and-var
- **Scoping rules –**
  - Variables declared by var keyword are scoped to the immediate function body (hence the function scope) while let variables are scoped to the immediate enclosing block denoted by { } (hence the block scope).
- **Hoisting –**
  - While variables declared with var keyword are hoisted (initialized with undefined before the code is run) which means they are accessible in their enclosing scope even before they are declared
  - let variables are not initialized until their definition is evaluated. Accessing them before the initialization results in a ReferenceError.
- **Creating global object property –**
  - At the top level, let, unlike var, does not create a property on the global object.
    ```
    var foo = "Foo";  // globally scoped
    let bar = "Bar"; // not allowed to be globally scoped
    console.log(window.foo); // Foo
    console.log(window.bar); // undefined
    ```
- **Redeclaration** –
  - In strict mode, var will let you re-declare the same variable in the same scope while let raises a SyntaxError.
    ```
    'use strict';
    var foo = "foo1";
    ```

```
var foo = "foo2"; // No problem, 'foo1' is replaced with 'foo2'.

let bar = "bar1";
let bar = "bar2"; // SyntaxError: Identifier 'bar' has already
been declared
```

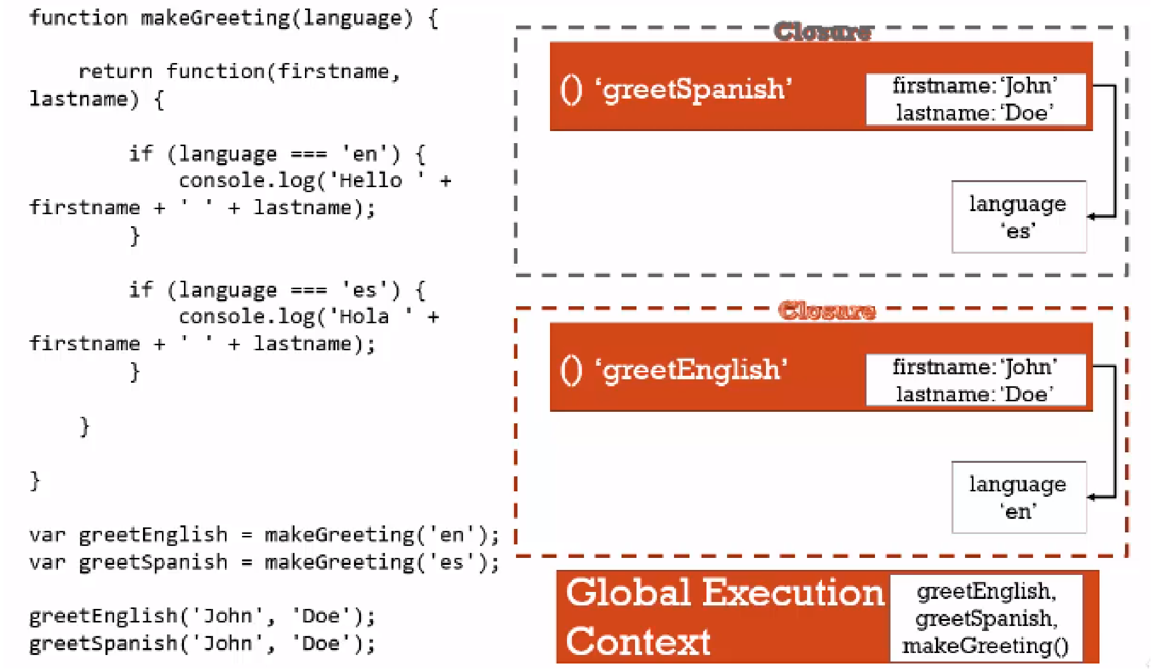
Framework Aside: Function Factories

- There are lots of ways in which closures are useful in JavaScript.
- We can use closures to our advantage for making patterns that would be otherwise impossible. E.g. Function Factories.
- A factory just means a function that returns or makes other things for me.
- A function factory just means a function that returns or makes other functions.
- Example – (28-function-factories)

```
/*
    There are lots of ways in which closures are useful in JavaScript.
    We can use closures to our advantage for making patterns that would be otherw
ise impossible.
    e.g. Function Factories.
*/
function makeGreeting(language) {

    return function(firstname, lastname) {
        if (language === 'en') {
            console.log('Hello ' + firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' + firstname + ' ' + lastname);
        }
    }
}
// seperate execution context for new call.
// So, greetEnglish is a function object whose closure points to language being E
nglish.
var greetEnglish = makeGreeting('en');

// seperate execution context for new call.
// And greetSpanish is a function object whose closure points to a different exec
ution context
// for the same function where language is Spanish.
var greetSpanish = makeGreeting('es');
```

```
greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```

- **Understanding the Flow -**
- It's important to remember that even though it's the same function (makeGreeting), every time I execute it, it creates a new execution context.
  It's a new memory space, no matter how many times I call it.
- The key is realizing that every time you call a function, it gets its own execution context, and any functions created inside of it will point to that execution context memory.



```
function makeGreeting(language) {

    return function(firstname,
lastname) {

        if (language === 'en') {
            console.log('Hello ' +
firstname + ' ' + lastname);
        }

        if (language === 'es') {
            console.log('Hola ' +
firstname + ' ' + lastname);
        }

    }

}

var greetEnglish = makeGreeting('en');
var greetSpanish = makeGreeting('es');

greetEnglish('John', 'Doe');
greetSpanish('John', 'Doe');
```

Closures and Callbacks

- A callback function, that's a function you give to another function to be run when the other function is finished.
- Example – [29-closures-and-callbacks]

```
function sayHiLater() {

    var greeting = 'Hi!';

    // the callback function is a function expression
    setTimeout(function() {

        // here 'greeting' is available due to closure
        // even though this callback functionruns 3 seconds
        // later sayHiLater() function finishes execution.
        console.log(greeting);

    }, 3000); // 3 seconds delay
}

sayHiLater();
```

call(), apply(), and bind()

- We have seen that "this" will be pointing at a different object or a different thing, depending on how the function is invoked. Refer – Objects, Functions, and 'this'
- Wouldn't it be nice to be able to control what the "this" variable ends up being when the execution context is created? Well, Java script has a way to do just that. And that's where call, apply and bind function come in.
- We already know a function is a special kind of object, it has a hidden optional name property which can be anonymous, we have a code property that contains the code and that code property is invokable so we can run the code.
- All functions in JavaScript also get access to some special functions, some special methods, on their own. (Remember, a function is just an object, so it can have properties and methods attached to it.)
- So, all functions have access to call(), apply() and bind() methods.
- All three of these methods have to do with the "this" variable and the arguments that you pass to the function as well.
- **bind() –**
    - Creates a copy of the function and the "this" variable inside that main function points to the argument passed to the bind() method.
- **call() –**
    - Unlike bind, which creates a copy of the function, call actually executes it.
    - call() method allows us to call the method normally.
      e.g. logName.call() is exactly same as logName()
    - Also we can pass an argument to it and then the 'this' variable inside that main function points to this argument.
      e.g. logName.call(person, 'en', 'es');
      Here the 'this' variable inside 'logName' will point to 'person' object.
    - We can also pass parameters which the main function accepts in the 2nd argument onwards.
    - e.g. logName.call(person, 'en', 'es');
      Here the 'this' variable inside 'logName' will point to 'person' object.

And arguments to logName: 'en', 'es'

- **apply() –**
  - apply() is exactly same as call() with only one difference.
  - The difference is apply() takes arguments as an array.
    e.g. logName.apply(person, ['en', 'es']);
- call, bind, apply are useful for **function borrowing**.
- bind() is useful for function currying as bind create a copy of function.
- **Function currying**: creating a copy of a function but with some preset parameters.
- This turns out to be really useful in mathematical situations. So if you're building a library that has to do a lot of mathematical calculations, you can have some fundamental functions that you can then build on with some other default parameters.
- IMP – Refer – 30-call-apply-bind-methods

Functional Programming

- Having **first class functions** in a JavaScript programming language means that we can implement what's called functional programming where we think and code in terms of functions.
- Instead of thinking purely in just separating your code into functions, you can start to think about how can we give our functions other functions or return functions from our functions, in order to even greater simplify the code that we're writing over and over again.
- Once you get used to it, it's very natural feeling to split things into functions, to pass them around to each other, because you're just splitting up the work in even finer more minute granular details.
- One other note about functional programming, your functions especially the tiny ones, as you're moving and passing little functions around that do work, should do their best **not to mutate data**. So it's always better to mutate data as high up in that chain of functions as possible, or better to not change it at all instead return something new.
- Example – 31-functional-programming

Open Source Example - Underscore.js

- Underscore.js library is a very famous library in JavaScript that helps you work with arrays and collections of objects.
- https://underscorejs.org/
- You can read the source code of such libraries and learn things for free.
- Open source libraries gives you open source education to be able to write better JavaScript, but also learn from really good JavaScript by being able to read good JavaScript.
- underscore.js implements a lot of functional programming concepts.

- Refer – 32-functional-programming-underscorejs

Object-Oriented JavaScript and Prototypal Inheritance

Conceptual Aside: Classical vs Prototypal Inheritance

- **Inheritance – one object gets access to the properties and methods of another object**.
- **Classical** Inheritance means the way it is done in other programming language like Java.
- **Prototypal** inheritance – something much simpler, it's very flexible, very extensible, and very easy to understand.
- Inheritance in JavaScript, is different compared with other programming languages.

Understanding the Prototype

- All objects In JavaScript (including functions) have a prototype property. This property is simply a reference to another object called proto (__proto__).
- Everything is an Object (or a primitive).
- Everything in JavaScript that isn't a primitive (number, string, boolean, etc.), so functions, arrays, basic objects - they all have a prototype (__proto__), except for the base object in JavaScript.
- Refer – 33-prototype

Reflection and Extend

- **Reflection**: an object can look at itself, listing and changing its properties and methods.
- So a JavaScript object has the ability to look at its own properties and methods. We can use that to implement a very useful pattern called **extend**.
- You can use reflection to have extend pattern (extend function in underscore.js library), not just the prototype pattern but to combine and compose objects.
- Refer – 34-reflection-and-extend

Building Objects

Function Constructors, 'new', and the History of Javascript

- Function constructor – A normal function that is used to construct objects.
- When you put that “new” keyword in front of a function call, the 'this' variable which is created during the creation phase of the execution context of that function, it points to a brand new empty object (created due to “new”). And that object is returned from the function automatically when the function finishes execution.
- The “new” operator makes the new empty object. And function constructors are used for adding properties and methods to that new object.
- So function constructors are used to set up properties and methods on new objects.
- IMP Refer – 35-function-constructors-new-and-history

Function Constructors and '.prototype'

- When you use a function constructor, it already set the prototype for you.
- Remember functions are special type of objects and they get some properties like name and code. Apart from these properties, there are some other properties which every function object gets called prototype. The prototype property starts off its life as an empty object, and unless you're using the function as a function constructor, it just hangs out, it's never used. But as soon as you use the new operator to invoke your function, then it means something.
- **The prototype property on a function is not the prototype of the function. It's the prototype of any objects created if you're using the function as a function constructor.**
- If you add any member (property or function) to the prototype property of a function, then that member (property or function) is available to ALL the objects created from that function (constructor).
- **It is always good practice to put your methods to prototype (of the function constructor)** because there will be only one instance of it (saves memory) and is accessible to all objects created from that function constructor. If you add methods to function constructor, those methods will be copied to each and every object created from that function constructor. This is not efficient because if you have say 100 objects, the methods created directly inside function constructor will get copied 100 times.
- From efficiency standpoint, it's better to put your methods on the prototype because they only need one copy to be used.
- Refer – 36-function-constructors-and-prototype

Dangerous Aside: 'new' and functions

- Remember function constructors are just another functions. So you could call it without using 'new' keyword in front of it, which is valid JavaScrit syntax.
- So if you forgot to use 'new' in front of function constructor, it will cause unintended side effects to your code and will be hard to debug.
- Function constructors are only there in the first place to try to appease syntactically programmers coming from other languages.
- Any function that we intend to be a function constructor, we should **capitalize** the first letter of its name so that you know that this function is intended to be "function constructor". This will help you to easily identify normal functions with function constructors.

Conceptual Aside: Built-In Function Constructors

- E.g. Number, String, etc.
  >var a = **new** Number(3);
  >a.toFixed(2); // toFixed() is available on Number.prototype
  "3.00"
- With built in function constructors (Number, String), it looks like you're creating primitives, but you're not. You're actually creating objects that contain primitives and give them extra abilities.
- There are cases where the JavaScript engine wraps up the primitive in its object for you, just so you can use properties and methods you might want to. (Won't convert number primitive to Number object automatically)
  E.g.
  > "sameer".length();
  Here the primitive string "sameer" is wrapped into String object and makes String.prototype methods (e.g. length()) available to use.
- This is somewhat useful, especially if you're building extra features in libraries or frameworks to tack on to these primitive values.
  - So you can add your custom properties/methods to built-in function constructors like Number, String.
  - With this, we can just enhance the JavaScript programming language just like that. And many libraries and frameworks use this technique to add features to add concepts and ideas and utilities.
  - **Be careful**, don't overwrite an existing or preexisting property or method of built-in function constructors.
- Refer – 37-builtin-function-constructors

Dangerous Aside: Built-In Function Constructors

- Built-in function constructors for primitive types, especially like boolean, number, string, are dangerous.
- By using built-in function constructors for creating primitives, you aren't really creating primitives. And strange things can happen during comparison with operators and coercion. It's better, **in general, to not use the built-in function constructors**.
- E.g.
```
>var a = 3
>var b = new Number(3);
>a == b
true
>a === b
False // because a and b are not of the same type
```
- If you're going to do a lot of date related work, use momentjs library, instead of a lot of work with built-in JavaScript date constructor.

Dangerous Aside: Arrays and for..in

- We can reference array elements using indexes .e.g 0, 1, 2 so arr[0]
- However these indexes are actually property names on the name-value pair.
- If you use for..in loop, it will print your array elements and it will also print any custom properties added to Array.prototype.
- In the case of arrays don't use for..in, use instead the standard for loop.
- Refer – 38-arrays-and-for-in

Object.create and Pure Prototypal Inheritance

- Function constructors were designed to **mimic** other languages that don't implement prototypical inheritance, and so they're a little awkward.
- Other languages implement classes, where a class defines what an object should look like and then you use the new keyword to create the object. And that's what function constructors are trying to mimic.
- On the other hand, many consider it better to simply focus on the fact that JavaScript does use prototypal inheritance, and not classical inheritance (like Java), and accept it, embrace it. And so yet another way to create objects that doesn't try to mimic other programming languages, and it's something that newer browsers all have built in. It's called Object.create.
- Object.create is not available on old browsers.
- A **polyfill** is code that adds a feature which the engine may lack.
- Object.create creates an empty object with its prototype pointing at whatever you passed into Object.create.

- This is pure prototypal inheritance. You simply make objects and then create new objects from them pointing to other objects as their prototype.
- So members (properties/methods) of Object.create() are **SHARED** among all the objects created from that object(e.g.person)
- If you want to define a new object, you create a new object that becomes the basis for all others. And then you simply override, hide properties and methods on those created objects by setting the values of those properties and methods on the new objects themselves.
- Refer – 39-Object.create-and-pure-prototypal-inheritance

ES6 and Classes

- A JavaScript class defines an object. A class in JavaScript is an object from which you create other objects.
- To set the prototype (__proto__), it uses keyword ‘extends’ to be used with class. And in the constructor, you need to call super() to simply call constructor of the super class.
- **Class in JS is just syntactic sugar (just another way to type your code). Understand the hood it’s still prototypal inheritance**.

Odds and Ends

'typeof' , 'instanceof', and Figuring Out What Something Is

- typeof and instanceof are operators but essentially functions under the hood.
- **typeof** operator tells us what you would expect, what type of thing is this.
- **instanceof** operator it tells if any object is down the prototype chain. If anywhere down that whole prototype chain, we find this type of object. And if I do, then this parameter is an instance of that one.
- Refer – 40-typeof-instanceof

Strict Mode

- JavaScript can be somewhat liberal about things when it comes to what it allows.
- There is a way that you to tell the JavaScript engine that you would like it to process your code in a stricter way, to be more regimented (very strictly organized or controlled.), to be pickier about what it lets you do.
- JavaScript is a very flexible language. And with flexibility, comes a lack of rules, sometimes. But strict mode can help you prevent errors under certain circumstances.
- To enable strict mode, add this line at top of your JS file or first line within your function (separate execution context in strict mode).
  "use strict";
- E.g. If you use strict mode, you are not allowed to use a variable without declaring.
- This is an extra feature. And not every JavaScript engine implements "use strict" the same way. They don't all agree on every rule that they'll implement. So this is an extra thing, and not something you can 100% rely on.
- **Gotcha** – if you have several JavaScript files, and then as part of your pushing them to production, you concatenate and minify them, that's very common where you put all of your JavaScript files together in one file so it only has to be downloaded once. And if that first file has "use strict" at the top, then the whole thing will be processed using that strict JavaScript engine flag. So you might cause yourself some trouble if other JavaScript files don't follow the strict rules. So **it's better to use "use strict" inside particular function only**.
- Refer – 41-strict-mode
- Strict Mode - MDN - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Examining Famous Frameworks and Libraries

Learning From Other's Good Code

- There are many aspects to improving as a programmer, as a developer. One of the most powerful is learning from others' good code.
- What's really terrific about working inside of the JavaScript architecture and the JavaScript community is that there's a fantastic treasure trove of good code out there for you to learn from.
- An Open Source Education.

Deep Dive into Source Code: jQuery

- Goals –
  - Able to read the source code.
  - See if we can learn from how it's structured, if we can gain some knowledge and some techniques, and borrow some ideas from inside source code.
- jQuery - make it easier to type syntactically certain things and it deals with cross browser issues, meaning that each browser, Firefox, Safari, Google Chrome, Internet Explorer, and various mobile versions of these browsers, all have their quirks and differences. jQuery handles those so you don't have to worry about it. You just are doing what you want to do and the code inside of jQuery is handling the browser quirks.
- jQuery essentially lets you manipulate the DOM.
- You can use $ sign or the word 'jquery'.
- **Method chaining**: calling one method after another, and each method affects the parent object.
  E.g. So obj.method1().method2() and both methods end up with "this" variable pointing at "obj".
- So all I have to do to make methods of an object chainable, is to finish the method, the last line of the method, is return the "this" variable.
  E.g. see jquery source code – addClass(), removeClass() methods. Thy return "this", which is the jquery object.
- **Don't be afraid….. of good code, it will make you a better JavaScript developer. ☺**
- Refer – 42-deep-dive-jquery

Let's Build a Framework / Library!

Requirements

- Greetr library
- Official requirements:
    - When I'm given a first name, a last name and an optional language, it should generate formal and informal greetings that I could use throughout my app.
    - It should support both English and Spanish languages for starters.
    - It should be reusable, meaning that it won't interfere with any of the other JavaScript code in my app, and someone else can just grab it and use it in their apps.
    - And lastly, we should have an easy to type structure, kind of jQuery like, something like **G$**.
    - greetr should also support jQuery. So even though it returns greetings what we'd like is to be able to give it a jQuery object that points at some HTML element. And it'll fill that element with the greeting.

Source Code of the Greetr library

- Refer – greetr-lib

Important Note

- **Put a semicolon before your IIFE**. That's another trick just in case there's some other code, some other library that may be injected before your code/library (Greetr.js) in a script file maybe above it, and if that doesn't quite finish out its semicolons properly.
- You can also put a semicolon there to make it more completely useful in that even if the other code doesn't finish its semicolons out properly (code that's used above your code), your code will still run fine.

TypeScript, ES6, and Transpiled Languages

- To **transpile** means to convert the syntax of one programming language to another.
- E.g. TypeScript, Traceur

ES6 Features with examples

- https://github.com/lukehoban/es6features

Tips and Tricks

- For our purposes, a framework and a library are the same thing, in the end it's just grouping of JavaScript code that performs a task and is intended to be reusable.
- An object is collection of name value pairs. And if the value is a primitive, it's called a **property**. And if the value a function, it's called a **method**.
- You can invoke a function statement before it's declared in the code (because of hoisting at execution context creation phase). However you cannot invoke a function expression before it's declared in the code (because function expressions are available only during execution phase of execution context).
- The prototype property on a function is not the prototype of the function. It's the prototype of any objects created if you're using the function as a function constructor.