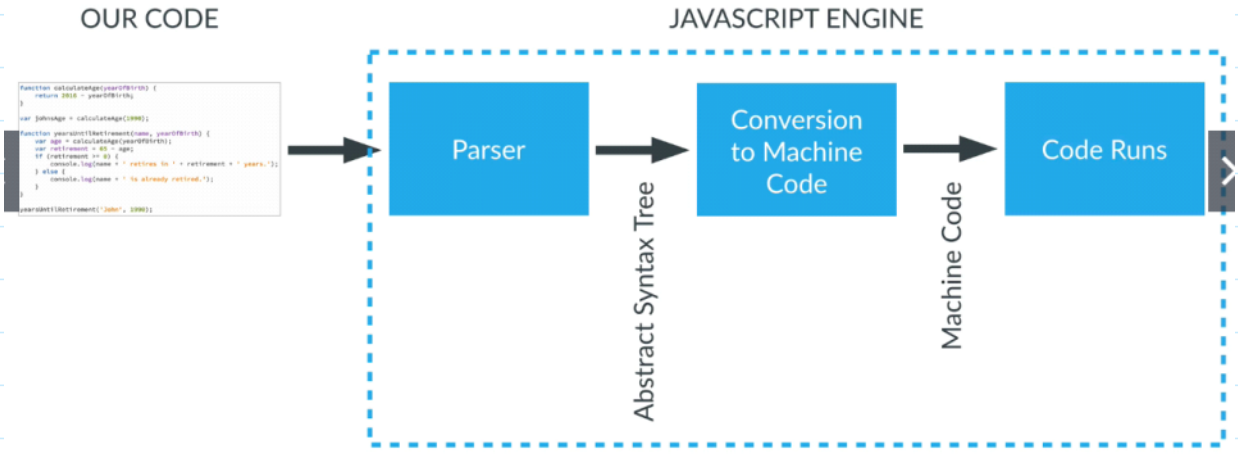
JavaScript Notes

Wednesday, May 6, 2020 2:27 PM

Resource Page: http://codingheroes.io/resources/

SECTION 3: How JavaScript Works Behind the Scene

36. How Code is Executed
    - JavaScript is usually hosted in an environment where it runs. This is typically a web Browser but it can also be a node.js server.
    - The code in which our code is hosted has some JavaScript Engine (a program that executes our code)
    - Inside the engine
        - There is a parser that reads the code line by line and checks the syntax validity
        - Meaning that the parser knows the JavaScript rules in order to alert the user of any mistakes and stops execution.
        - If everything is correct then the parser produces a data structure called *Abstract Syntax Tree* which is then translated to machine code
        - The machine code can then be read by the computer processor and the code runs

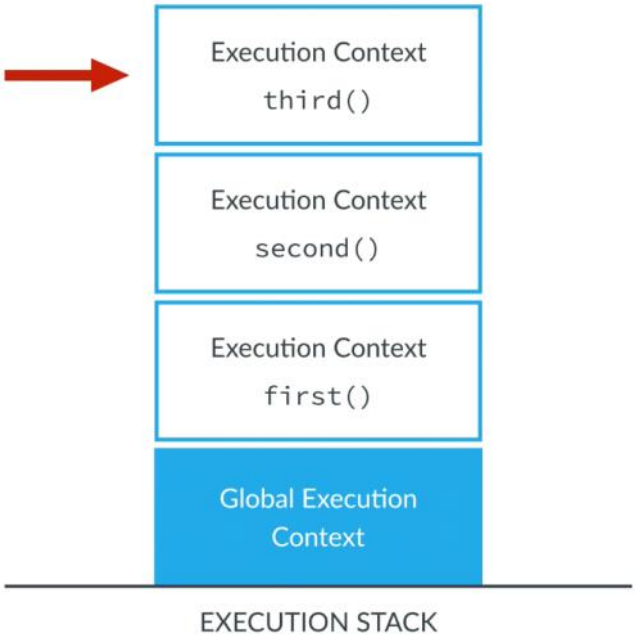


37. Execution Context and Execution Stack
    - All JavaScript code needs to run in an environment. These environment are called execution context.
    - These execution context can be thought of as a box which stores variables and has a piece of our code evaluated and executed
    - The default execution context is called the *Global Execution Context*.

- Stores variables and pieces of code that are not in a function

- Every time we call a function, it gets its own execution context and is pushed on to the execution stack. Once the function returns, then they are popped off the execution stack

```
var name = 'John';

function first() {
    var a = 'Hello!';
    second();
    var x = a + name;
}

function second() {
    var b = 'Hi!';
    third();
    var z = b + name;
}

function third() {
    var c = 'Hey!';
    var z = c + name;
}

first();
```



38. Execution Contexts in Detail: Creation and Execution Phases and Hoisting
    - We can associate an execution context (EC) as an object that has 3 properties
        i. Variable Object (VO)
        ii. Scope chain
        iii. 'This' variable
    - When a function I called, a new EC is placed on top of the stack. This happens in two phases:
        1. Creation phase : where the properties of the EC object are defined
            a) Creation of the VO
                - Argument object is created that contains all the arguments passed into the function
                - Code is scanned for function declarations. For each function declared, a property in the VO is created pointing to the function.
                - Code is scanned for variable declaration. For each variable, a property is created in the VO set to undefined (will only be defined in the execution phase).

                HOISTING

            b) Creation of the scope chain
            c) Determines the value of the this object
        2. Execution phase : the code that generated the EC is ran line by line

39. Hoisting in Practice
    - Note that hoisting only works with function declaration and not function

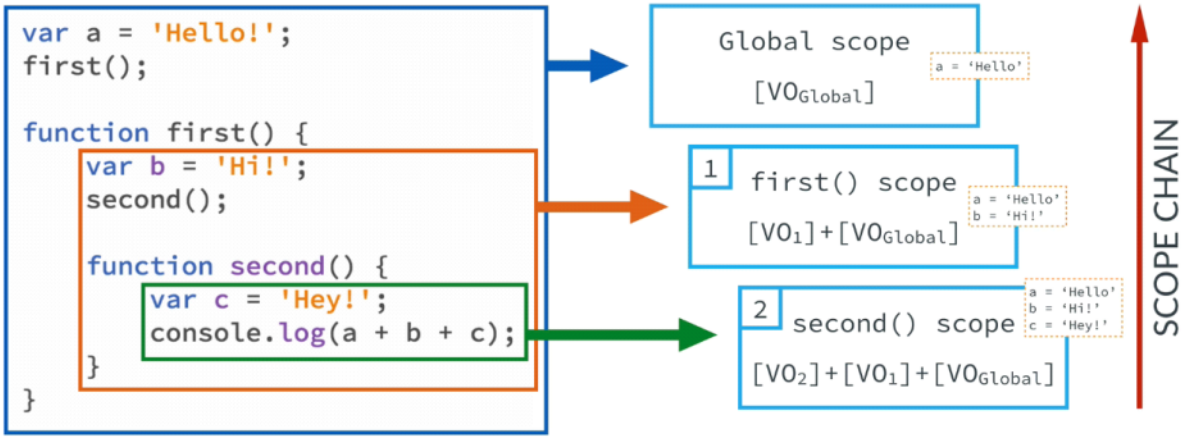
expression. This is because when the function is called, EC is placed on to the stack and a VO is created in which the code is scanned for *function declaration* and not function expression

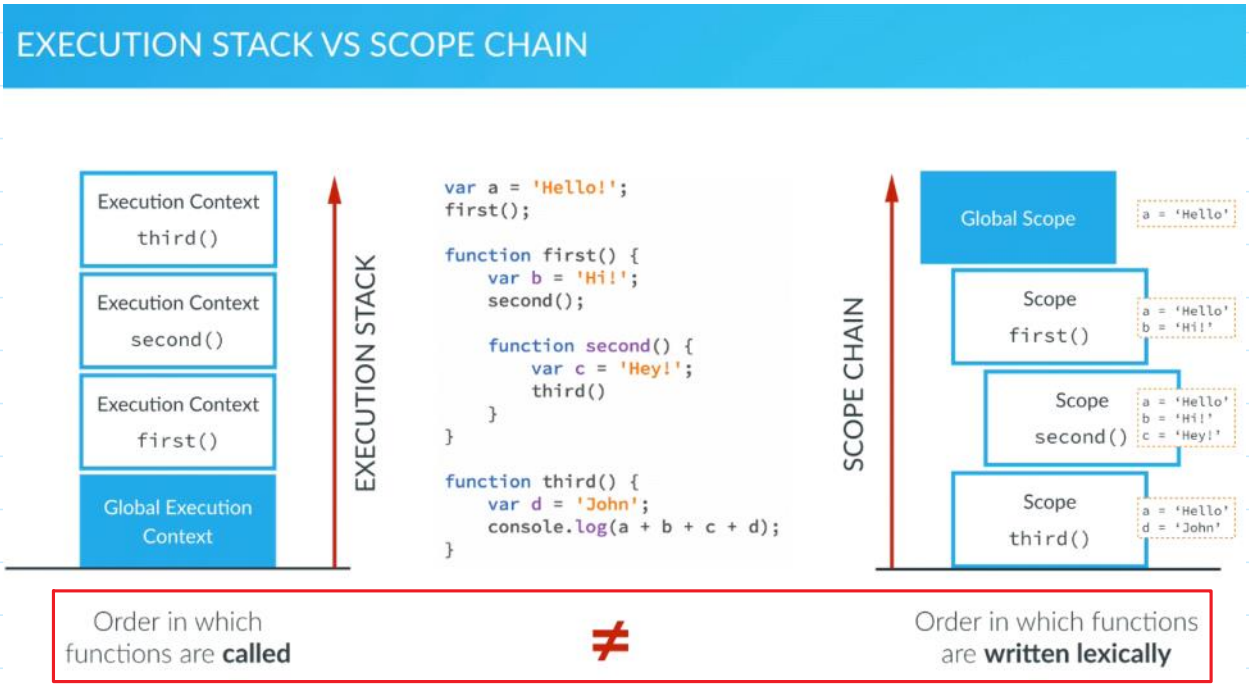
```
// functions
calculateAge(1996); // function call works before or after declaration

// hoisting only works with function declaration
function calculateAge(year) {
    console.log(2016 - year);
}

calculateAge(1996); //  function call works before or after declaration


//retirement(1990); // error

// function expression does not hoist
var retirement = function(year) {
    console.log(65 - (2016 - year));
}

retirement(1996); // works only after the function has been expressed


// variables
console.log(age); // undefined
var age = 26;
console.log(age); // variable defined in global execution context

function foo() {
    console.log(age); //undefined
    var age = 65;
    console.log(age); // variable defined in a a different execution context
}

foo();
console.log(age);
```

```
20
20
45
```

```
undefined
26
undefined
65
26
```

40. Scoping and the Scope Chain
    - Scoping answers the question "Where can I access a certain variable"
    - Each new function creates a new scope (a space/environment where the variable it defines are accessible)
        - In other langunage a scope is also created by if, while, or for blocks, but **NOT** in JavaScript. The only way a new scope is created is when a new function is created
    - *Lexical Scoping:* when a local scope includes the scope of its parent function
        - Note: local scopes are not visible to its parent or global scopes (does not work backwards)





41. 'This' Keyword
    - As we already know, the 'this' variable is stored in the EC object, meaning that a function or method has been called, and an EC has been placed on the stack.
    - Now where does the 'this' keyword point to depending on the call?
    - In a **regular function call**, the 'this' key work **points at global object**
    - In a **method call**, the 'this' variable **points to the object calling the method**
    - The 'this' keyword is not assigned a value until a function where it is defined is called

42. 'This' Keyword in Practice
    - The 'this' keyword in this context was called from a regular function call, meaning that the 'this' keyword will point to the global object, which in this case is the window object

```
calculateAge(1996);

function calculateAge(year) {
    console.log(2016 - year);
    console.log(this); // still resides in window obj
}
```

```
20
▶ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
```

- The first 'this' keyword in this context was called from a method call and therefore will point to the object in which called the method. In this case the object john called the method calculateAge that includes a 'this' variable, meaning that the 'this' variable will point to the john object
- The second 'this' keyword from the innerfunction() DESPITE being inside an object is a regular function call which makes the 'this' keyword point to the global object.

```
var john = {
    name: 'John',
    yearOfBirth: 1990,
    calculateAge: function() {
        console.log(this); // this belong to john obj
        console.log(2016 - this.yearOfBirth);

        function innerFunction() {
            console.log(this); //regular function call, point to window obj
        }

        innerFunction();
    }
}

john.calculateAge();
```

```
▶ {name: "John", yearOfBirth: 1990, calculateAge: ƒ}
26
▶ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
```

- Below is a example of an important behavior of the 'this' variable. We see from below that the object mike calls the calculateAge() function borrowed from the john object and calculates mike age. This proves that the 'this' variable is not assigned a value until the function in which is it defined is called.

```
var john = {
    name: 'John',
    yearOfBirth: 1990,
    calculateAge: function() {
        console.log(this); // this belong to john obj
        console.log(2016 - this.yearOfBirth);
    }
}

john.calculateAge();

var mike = {
    name: 'Mike',
    yearOfBirth: 1984
};

// Method borrowing
mike.calculateAge = john.calculateAge;

mike.calculateAge();
```

```
▶ {name: "John", yearOfBirth: 1990, calculateAge: ƒ}
26
▶ {name: "Mike", yearOfBirth: 1984, calculateAge: ƒ}
32
```

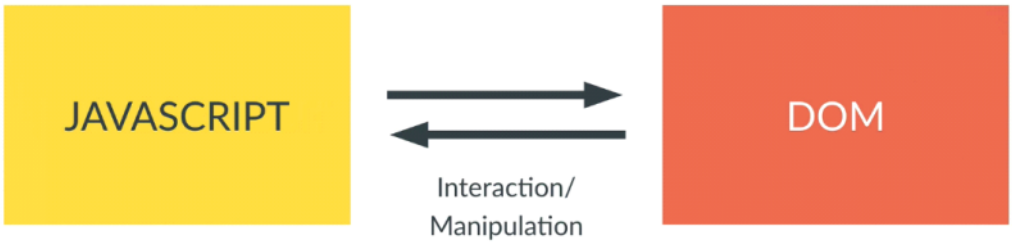
Note that here, the function is not being called and is treated similar to a variable. That is why the () characters are not included after the function name. Only when a function is CALLED will the parentheses () e necessary

SECTION 4: JavaScript in the Browser: DOM Manipulation and Events

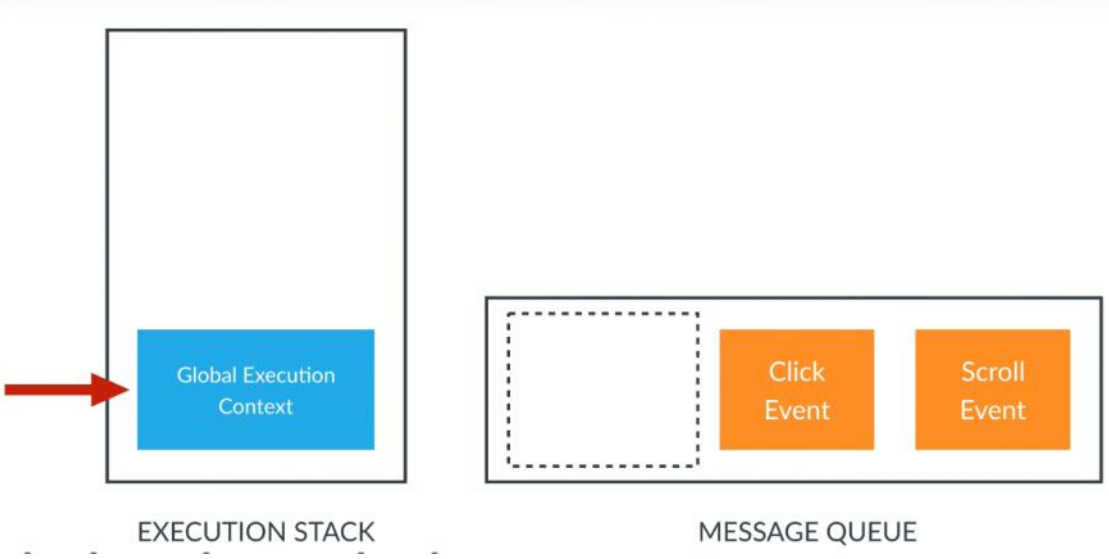
45. The DOM and DOM Manipulation
    - DOM: Document Object Model
        - It is a structured representation of an HTML Document
        - It is used to connect webpages to scripts like JavaScript
        - For each HTML box, there is an object in the DOM that we can interact with

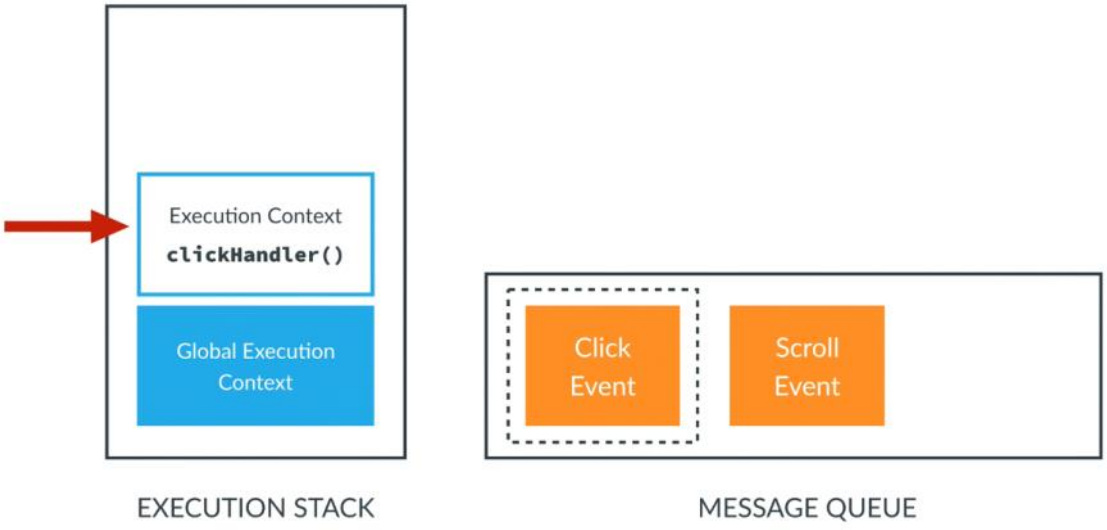
```
<body>
    <section>
        <p>A paragraph with a <a>link</a>.</p>
        <p>Another second paragraph.</p>
    </section>
    <section>
        <img src="x.jpg" alt="The DOM">
    </section>
</body>
```

- Note that: JavaScript and the DOM are two different things. We can use **JavaScript methods** that can interact with the DOM which means the webpage



49. Events and Event Handling
    - *Events*: notifications sent to the code to let it know that something happened to the webpage (ie. Click, scroll, hover)
    - *Event Listener:* Function that performs an action based on a certain event. It waits for a specific event to happen
    - **How events are processed?**
        - The rule: Event can only be processed when all the EC in the execution stack is empty.
        - *Message Queue:* where events are put, sitting there waiting to be processed which only happens when the execution stack is empty
        - Since, event listeners are function, when they are called (when an event has occurred) event listener gets its own EC which then gets put on the stack and becomes the active EC.





- *Call Back Function:* function that we pass as an argument into another function

```
document.querySelector('.btn-new').addEventListener('click', init);

function init() {
    // Do something here
}
```

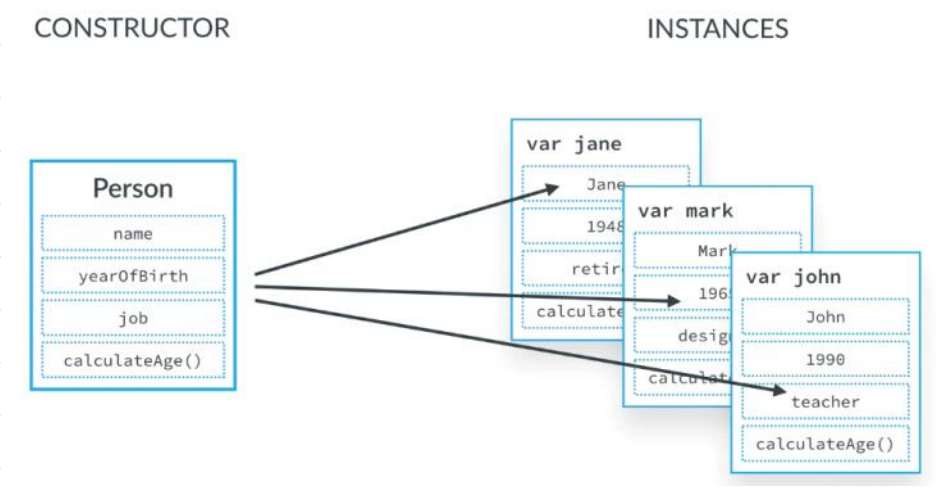
- *Anonymous function:* function that doesn't have a name and therefore cannot be reused

```
document.querySelector('.btn-hold').addEventListener('click', function () {
    // Do Something here
})
```

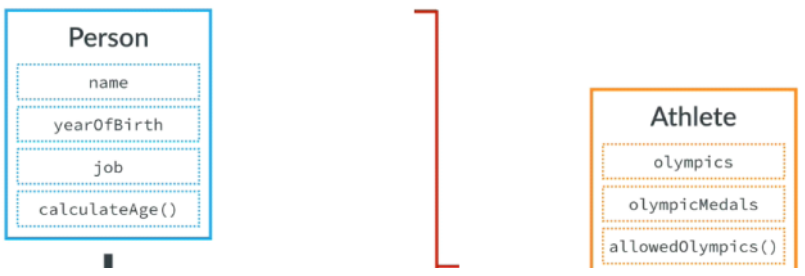
SECTION 5: Advanced JavaScript: Objects and Functions

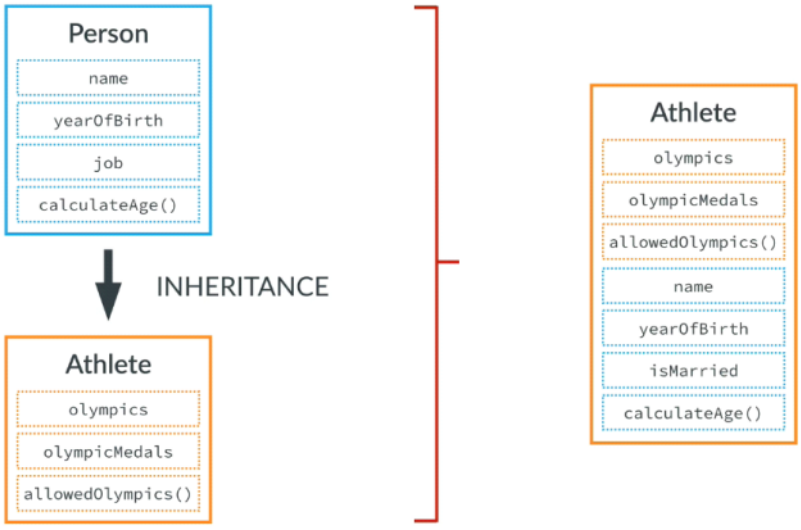
60. Everything is an Object: Inheritance and the Prototype Chain

- Object Oriented Programming (OOP)
  - Objects interact with one another through methods and properties
  - Keeps code clean, sued to store data, structure applications into modules
  - In JavaScript a *class* is called *constructor* or *prototype* which acts as a blueprint. Based on this constructor, we can create as many instances of itself or objects as we want.

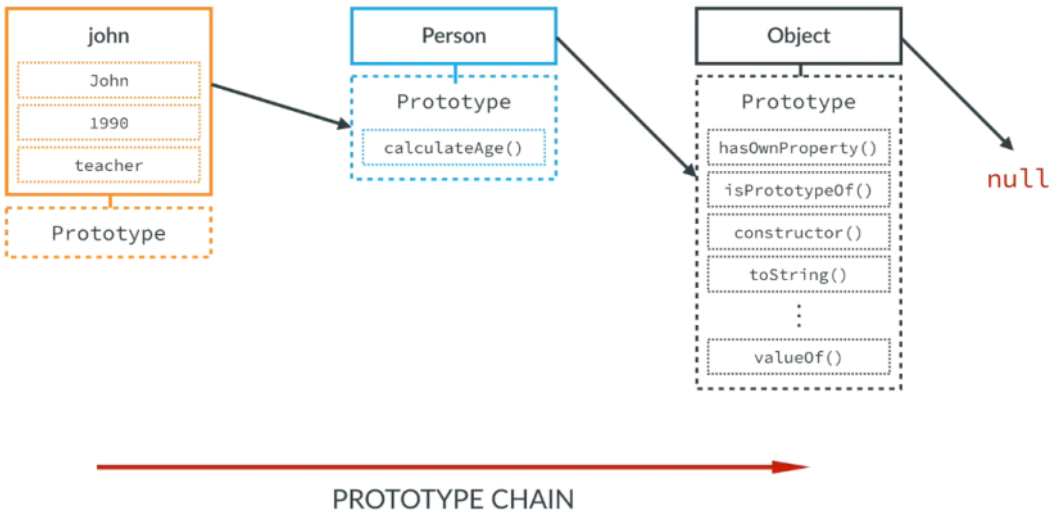


- Inheritance in General
  - When an object is based on another 'parent' object. This object inherits the properties and methods of the parent object.





- Every JavaScript object has a prototype object, which makes inheritance possible in JavaScript
- The prototype property of an object is where we put methods and properties we want other objects to inherit
- The constructor's prototype property is NOT the prototype of the constructor itself. It is the prototype of ALL the instances hat are created through it



61. Creating Objects: Function Constructors

- Object literal

```
var john = {
    name: 'John',
    yearOfBirth: 1996,
    job: 'teaacher'
};
```

We pass in arguments that we will set with our object

```
    job: 'teacher'
};
```

- Function Constructor
  - Pattern for writing a blueprint
  - **Convention:** name should be capitalized

```
var Person = function(name, yearOfBirth, job) {
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
    this.calculateAge = function() {
        console.log(2020 - this.yearOfBirth);
    }
}
```

The 'this' operator will set the arguments passed in the function as the new object's properties

- Once we have a constructor in place, we can now create instances of the constructor called objects. This is called *instantiation*
- The *new* operator
  - When the *new* operator is used a new *EMPTY object* is created
  - Then the function is called
  - Calling a function created a new EC in the stack. Recall that a regular function call results in the 'this' variable to point to the global object. HOWEVER, because the *new* operator first created an empty object that called this function, the 'this' variable of that function point the new object created by the *new* operator.
  - That new empty object is then assigned to the john object :)

```
var john = new Person('John', 1996, 'teacher');
john.calculateAge();
```

- Methods in constructors
  - In the current example it is viable to have the calculateAge() method inside the constructor. But what if we have over 20 method each with over 100 lines of code inside the constructor? That would not be very efficient nor easy to follow
  - An alternative way of writing methods for constructors is the following:

```
Person.prototype.calculateAge = function() {
    console.log(2020 - this.yearOfBirth);
}
```

  - Recall that the constructor has a prototype property in which I inherits from. Now we can use a method that is not necessarily inside the function constructor
  - This method can also be used for variables, but it is not a common practice:

```
Person.prototype.numberOfLegs = 2;
```

62. The Prototype Chain in the Console

- Using the console, we can inspect objects in our code, its properties, variables, and inheritance, more specifically, the console displays the prototype chain in a clear way

```
> john
< ▼Person {name: "John", yearOfBirth: 1996, job: "teacher"}
    job: "teacher"
    name: "John"
    yearOfBirth: 1996
  ▼__proto__:
    ▶calculateAge: ƒ ()
     type: "human"
    ▶constructor: ƒ (name, yearOfBirth, job)
    ▼__proto__:
      ▶constructor: ƒ Object()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
```

- With visibility on the Object object methods, because of inheritance, we can call these methods from the john object to retrieve more information about the john object as follows:

```
> john.__proto__ === Person.prototype
< true
> john.hasOwnProperty('name')
< true
> john.hasOwnProperty('talent')
< false
> john instanceof Person
< true
>
```

- Can also use console.info([data]) to inspect objects that may not display their properties or prototype chain right away

```
> var x = [4, 7, 3]
< undefined
> console.info(x)
  ▼(3) [4, 7, 3]
     0: 4
     1: 7
     2: 3
     length: 3
    ▶__proto__: Array(0)
< undefined
>
```

- This further proves that mostly everything in JavaScript is an object

63. Creating Objects: Object.create
    - Here we will talk about objects that inherit from a prototype instead of a constructor
    - In this case, we first defined an object that will act as a prototype and then create an object based on that prototype

```
var personProto = {
    calculateAge: function() {
        console.log(2016 - this.yearOfBirth);
    }
};

var john = Object.create(personProto);
john.name = 'John';
john.yearOfBirth = 1998;
john.job = 'teacher';

var jane = Object.create(personProto,
    {
        name: { value: 'Jane' },
        yearOfBirth: { value: 1969 },
        job: { value: 'designer' }
    });
```

    - The main difference with using Object.create is that the new object inherits directly from the argument passed (ie. personProto object) rather than inheriting from the constructor's prototype property from function constructors.
    - **Advantage of Object.create**: it allows easy implementation of complex inheritance structures. It allows the direct specification of which object should be the prototype
    - Function constructors and Object.create are the most popular methods for inheritance

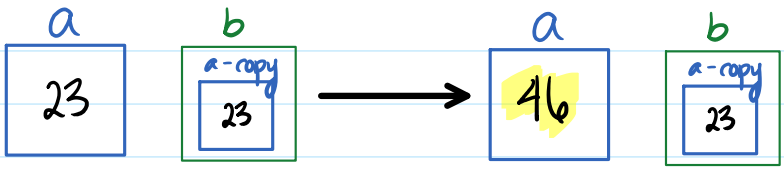
64. Primitives vs. Objects
    - Primitives hold value of that variable
    - Object points to the object

    - Primitives
        - Observe the code below along with its output

```
var a = 23;
var b = a;
a = 46;
console.log(a);
console.log(b);
```

```
46
23
```

- Notice that the variable a has changed to 46 but variable b remains as 23
- This is because when we assigned variable b it holds their own copy of the data and do not reference anything. When variable a changes, variable b is unaffected

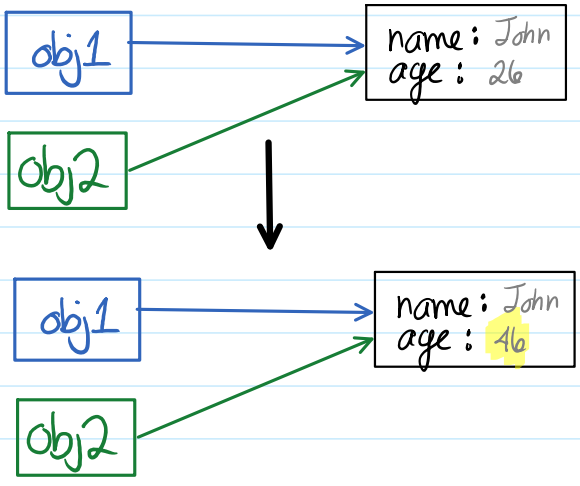


- Objects
  - Notice that when the object property obj1.age changed so did the object property obj2.age

```
var obj1 = {
    name: 'John',
    age: 26
};
var obj2 = obj1;
obj1.age = 46;
console.log(obj1);
console.log(obj2);
```

```
{name: "John", age: 46}
    age: 46
    name: "John"
  ▶ __proto__: Object
{name: "John", age: 46}
    age: 46
    name: "John"
  ▶ __proto__: Object
```

  - This is because when an object is created it points to a reference that holds that object. And thus when obj1 changes, obj2 which also point to the same object as obj1 will change accordingly



- It is important to understand this as it can cause some confusion and bugs in codes when used in functions.
- Below is an example of how primitives and objects differ in behavior when attempting to change their value or property using a function

```
var age = 23;
var obj = {
    name: 'Jonas',
    city: 'Lisbon'
};

function change(a, b) {
    a = 46;
    b.city = 'San Francisco';
}

change(age, obj);
console.log(age);
console.log(obj.city);
```

```
23
San Francisco
```

- Notice how the function change, attempted to change the primitive variable age but was not able to. However, it successfully changed the object property. This result checks out with what we just learned about primitive and object.
- This proves that when a primitive is passed into the function, a simple copy is created. But when we passed the object, we actually pass the reference of the object

65. <u>First Class Functions: Passing Functions as Arguments</u>
- Recall that a function is an instance of the Object type
- A function behaves like any other object
- We can store function is a variable
- We can pass functions as an argument to another function
- We can return a function from a function

- **Call back function**
    - When we pass a function to another function, we don't include the parenthesis because including the parenthesis will call the function right away, but that is not what we want
    - We want the function that is passed in as an argument to be called later and used inside the other function

```
var years = [1990, 1995, 1937, 2005, 1998];

function arrayCalc(arr, fn) {
    var arrRes = [];
    for (var i = 0; i < arr.length; i++) {
        arrRes.push(fn(arr[i]));
    }
    return arrRes;
}

function calculateAge(el) {
    return (2016 - el);
}

function isFullAge(el) {
    return (el >= 18);
}

function maxHeartRate(el) {
    if (el >=18 && el <=81) {
        return Math.round(206.9 -(0.67 * el));
    } else {
        return -1;
    }

}

//arraycalc(years, calculateAge());

// we dont want to call the calculateAge function here
// we want to call it later to be used inside the arraCalc function
var ages = arrayCalc(years, calculateAge);
console.log(ages);

var fullAges = arrayCalc(ages, isFullAge);
console.log(fullAges);

var rates = arrayCalc(ages, maxHeartRate);
console.log(rates);
```

```
▼ (5) [26, 21, 79, 11, 18]
    0: 26
    1: 21
    2: 79
    3: 11
    4: 18
    length: 5
  ▶ __proto__: Array(0)
▼ (5) [true, true, true, false, true]
    0: true
    1: true
    2: true
    3: false
    4: true
    length: 5
  ▶ __proto__: Array(0)
▼ (5) [189, 193, 154, -1, 195]
    0: 189
    1: 193
    2: 154
    3: -1
    4: 195
    length: 5
  ▶ __proto__: Array(0)
```

66. First Class Functions: Functions Returning Functions
    - When we are returning a function from a function, we are essentially returning an object that happens to be a function, since functions in JavaScript are just objects

```
function interviewQuestion(job) {
    if (job == 'designer') {
        return function(name) { // Anonymous function
            console.log(name + ', can you please explain what UX design is?');
        }
    } else if (job == 'teacher') {
        return function(name) {
            console.log('What subject do you teach, ' + name + '?');
        }
    } else {
        return function(name) {
            console.log('Hello ' + name + ', what do you do?');
        }
    }
}
```

- Using the above code into action, when we assign the 'teacherQuestion' variable to the 'interviewQuestion' function with the 'teacher' argument, then the variable will be the function in which a teacher interview question will be asked.
- The 'teacherQuestion' variable will now be a function that takes in a 'name' parameter

```
var teacherQuestion = interviewQuestion('teacher');
var designerQuestion = interviewQuestion('designer');

teacherQuestion('John');
designerQuestion('Mary');
designerQuestion('Jane');
teacherQuestion('Mike');

// Another way of doing this is the following:
interviewQuestion('teacher')('Mark');
```

```
What subject do you teach, John?
Mary, can you please explain what UX design is?
Jane, can you please explain what UX design is?
What subject do you teach, Mike?
What subject do you teach, Mark?
```

- ** With this method, we can write one generic function and then create a bunch more specific function based on that generic function **

66. <u>Immediately Invoked Function Expressions (IIFE)</u>
    - Let's say we want to create a game in which a random number is generated between 0-10. If the outcome is less than 5 the player loses if greater, the player wins.
    - Important thing to note is that we want the generated number to be hidden
    - One way this could be done is the following

```
function game() {
    var score = Math.random() * 10;
    console.log(score >= 5);
}

game();
```

- However, this method causes some problems and can be done a different better way using **IIFE**
- In high level, what we want is to create a new scope that is hidden from the outside scope. Where we can safely put variables. With this we can obtain privacy without interfering with other variables in the other scopes. IIFE allows us to do exactly that
- The below code is how IIFE is implemented in JavaScript

```
(function (goodluck) {
    var score = Math.random() * 10;
    console.log(score >= 5 - goodluck);
})(4);
```

- How this works
  - Recall that what is inside parenthesis cannot be a statement.
  - By writing the function inside the parenthesis, we trick JavaScript to treat this as an EXPRESSION and NOT A DECLARATION.
  - Afterwards, we have to invoke the function by adding the parenthesis '()' right after.
  - Also, we can pass arguments to the IIFE.

- IIFE allows code privacy and modularity

68. Closures

- Closures is when an inner function always has access to the variables and parameters of its outer function, EVEN AFTER THE OUTER FUNCTION HAS RETURNED.
- Lets take a closure look at how this happens
  - When the retirement()is called a new EC is put on the stack
  - and a new box for the retirement scope has been added to the chain that points to the variables of the retirement()

- Once the retirement() returns, the EC is popped off the stack along with its VO and the entire scope chain should be gone, right? ACTUALLY NO.
- ***The VO from the EC is still there and <u>sits in memory and can be accessed.</u>***
- Note that the scope chain is a pointer to the VO and is still there as well.
- Next, when the retirementUS() function is called, a new EC for this function is pushed on to the stack
- Since the retirementUS() is written lexically in the retirement(), it has access to the retirement() scope.
- Since, the VO of the outer function is still there, the scope chain remains intact and allows us to have access to the scope of the outer function.

- The power of closures in application: Implement the interviewQuestion() written earlier but using the power of closures

```
function interviewQuestion(job) {
    return function(name) {
        if (job == 'designer') {
            console.log(name + ', can you please explain what UX design is?');
        } else if (job == 'teacher') {
            console.log('What subject do you teach, ' + name + '?');
        } else {
            console.log('Hello ' + name + ', what do you do?');
        }
    }
}

var teacherQuestion = interviewQuestion('teacher');
```

```
function interviewQuestion(job) {
    return function(name) {
        if (job == 'designer') {
            console.log(name + ', can you please explain what UX design is?');
        } else if (job == 'teacher') {
            console.log('What subject do you teach, ' + name + '?');
        } else {
            console.log('Hello ' + name + ', what do you do?');
        }
    }
}

var teacherQuestion = interviewQuestion('teacher');
var designerQuestion = interviewQuestion('designer');

teacherQuestion('John');
designerQuestion('Mary');
designerQuestion('Jane');
teacherQuestion('Mike');
```

- Using the power of closure, we were able to create cleaner code the inner function still had access to the job variable and was able to use this information to make decisions
- Essentially all the work is handled in the inner function

69. Bind, Call, and Apply

- We know that functions are essentially objects. And just like the array object, the function object has methods of its own that we can use : bind, call, apply
- These allows us to call the 'this' variable and set it manually.
- Consider the following objects written below

```
var john = {
    name: 'John',
    age: 26,
    job: 'teacher',
    presentation: function(style, timeOfDay) {
        if (style === 'formal') {
            console.log('Good ' + timeOfDay + ' ladies and gentlemen! I\'m '
            + this.name + '. I\'m a ' + this.job + ', and I\'m ' + this.age
            + ' years old.');
        } else if (style === 'friendly') {
            console.log('Hey! What\'s up? I\'m ' + this.name + ' I\'m a ' +
            this.job + ', and I\'m ' + this.age + ' years old. Have a nice '
            + timeOfDay + '.');
        }
    }
};

var emily = {
    name: 'Emily',
    age: 35,
    job: 'designer'
};
```

- Call
  - Also known as *method borrowing*.
  - This allows us to set the 'this' variables to point to another object, and allow that object to use a method from a different object but with its own variable associates
  - The first arguments will set the 'this' variable to point to the object passed.

```
var emily = {
    name: 'Emily',
    age: 35,
    job: 'designer'
};

john.presentation('formal', 'morning');
john.presentation.call(emily, 'friendly', 'afternoon');
```

```
Good morning ladies and gentlemen! I'm John. I'm a teacher, and I'm 26 years old.
Hey! What's up? I'm Emily I'm a designer, and I'm 35 years old. Have a nice afternoon.
```

- Apply
  - Similar to the call method except it uses an array to set the arguments.

```
john.presentation.apply(emily, ['friendly', 'afternoon']);
```

- Bind
  - Also known as *carrying*
  - Bind doesn't immediately call the function but instead generates a copy of the function, so that we can store it somewhere. This can be extremely useful when dealing with preset arguments.
  - The Bind method allows us to create a function with preset arguments. This allows for some functions to be categorized.

```
var johnFriendly = john.presentation.bind(john, 'friendly');

johnFriendly('morning');
johnFriendly('night');

var emilyFormal = john.presentation.bind(emily, 'formal');

emilyFormal('afternoon');
emilyFormal('night');
```

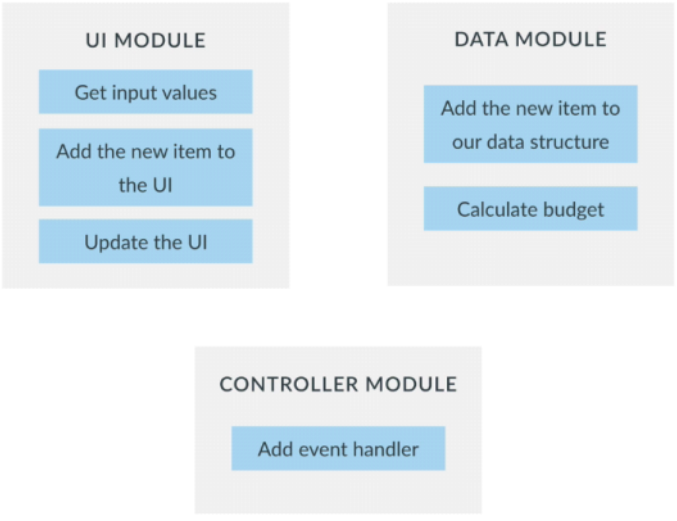
```
Hey! What's up? I'm John I'm a teacher, and I'm 26 years old. Have a nice morning.
Hey! What's up? I'm John I'm a teacher, and I'm 26 years old. Have a nice night.
Good afternoon ladies and gentlemen! I'm Emily. I'm a designer, and I'm 35 years old.
Good night ladies and gentlemen! I'm Emily. I'm a designer, and I'm 35 years old.
```

SECTION 6: Putting It All Together: The Budget App Project

76. Project Planning and Architecture: Step 1
- Todo List (part 1)
  - Add event Handler
  - Get input values
  - Add new item to our data structure
  - Add new item to the UI
  - Calculate budget
  - Update UI

- Structuring our code
  - Modules
    - It is an important aspect of any *robust* applications architecture
    - Keep code cleanly separated and organized
    - Encapsulate some data into privacy and expose other data publicly
    - Allows us to break our code into separate parts and allows them to interact with each other creating a working system

- Below is how we have delegated our todo list into modules
  - The controller module will be act as the central point for task delegation and action



77. Implementing the Module Pattern
- Recall that we want: we want to keep sections of our code that are related with one another inside of independent and organized units: modules.
- *Data Encapsulation*
  - Allows us to hide the implementation details of a specific modules to the outside

scope so that we only expose a public interface. Which is also known as ***API***
    - Each of these modules will have variables and functions that are private (only accessible inside of the module meaning that no other modules can overwrite these data) and public (other modules can have access to these data and can interact with it)

- The secret of the module pattern is that it returns an object that has all the public function from that module. These are all the function that we want the outside scope to have access to.

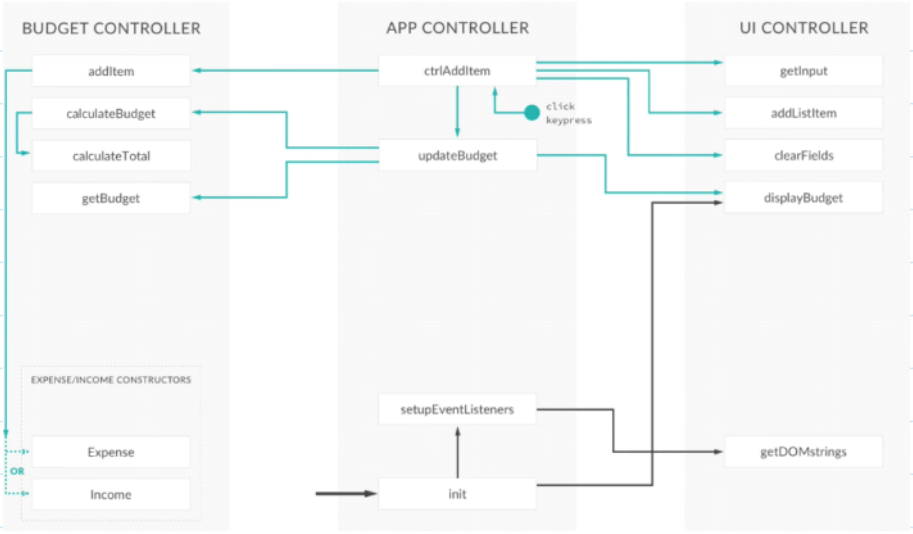
- These can be done through IIFEs
- Notice how the controller module takes in two parameters: budgetCtrl and the UICtrl

```
var budgetController = (function () {

    var x = 23;

    var add = function(a) {
        return x + a;
    }

    return {
        publicTest: function(b) {
            return (add(b));
        }
    }

})()
```

```
var UIController = (function () {

    // some code

})();
```

```
var controller = (function(budgetCtrl, UICtrl) {

    var z = budgetCtrl.publicTest(5);

    return {
        anotherPublic: function() {
            return z;
        }
    }

})(budgetController, UIController);
```

88. Project Planning and Architecture: Step 2
    - Below shows a high level overview of the functionality of our modules

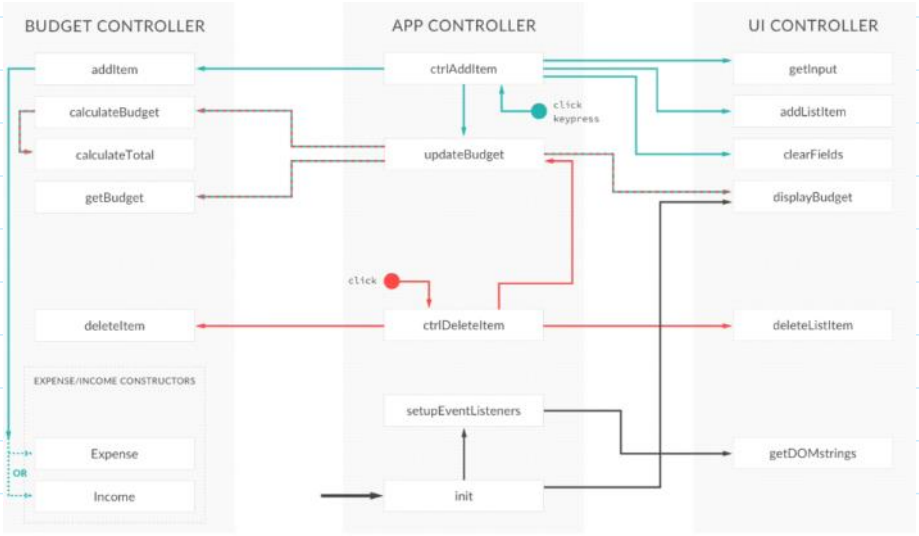


- Todo List (part 2)
  - Add event handler (to delete)
  - Delete the item from the data structure
  - Delete the item from the UI
  - Re calculate the budget
  - Update the UI

89. Event Delegation
- *Event Bubbling:* This is when an event is triggered on an element, then the exact same event is also triggered in all of the parent elements one at a time, all the way up to the DOM tree
- *Target Element:* the element in which the event is fired from
- *Event Delegation:* Technique in which we attach an event handler to a parent element of the target element. Once the event is fired we wait until it bubbles up the parent element with the event handler attach, and now we can do what we intended to do to the target element.

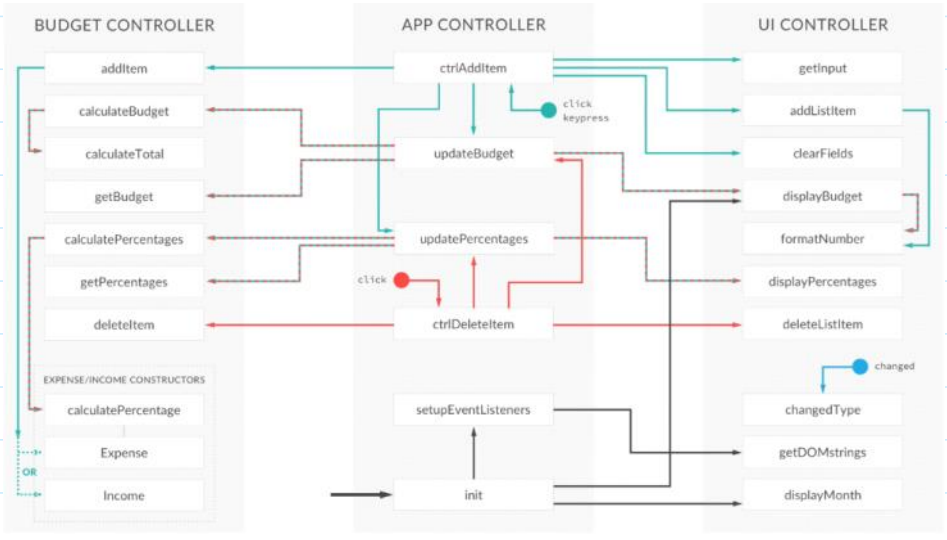
- Use Cases for Event Delegation
  1. When we have an element that has lots of child elements that we are interested in. In this case, we can attach an event handler to the parent element and then determine in which element that event was fired
  2. When we want an event handler attached to an element that is not yet in the DOM when our page is loaded. That is because we cannot add an event handler to something that is not in our page

93. Project Planning and Architecture: Step 3
- Below is a high level overview of the updated functionality of our modules



- Todo List (part 3):
  - Calculate percentages
  - Update percentage in UI
  - Display correct month and year
  - Number formatting
  - Improve input field UX

100. <u>Final Considerations</u>
- Our final architecture



SECTION 7: Next Generation JavaScript: Intro to ES6 / ES2015

103. What is New in ES6 / 2015
- Variable declaration with let and const
- Blocks and IIFEs
- Strings
- Arrow Functions
- Destructuring
- Arrays
- The Spread Operator
- Rest and Default Parameters
- Maps
- Classes and subclasses
- Promises
- Native Modules

104. Variable declaration with let and const
- *Let*
  - Is similar to var. The variables that can be changed or mutated
  - Blocked scoped
  - Throws an error when variable is not defined
- *Const*
  - Used for variables that cannot be changed or mutated
  - Block scoped
  - Variable has to be defined upon declaration
- *Var*
  - Used for variables that can be changed or mutated
  - Function scoped
  - Variable can still be used even if undefined

1. Example of variable mutation in *const* declaration vs *let* declaration vs *var* declaration
   - The 'name6' variable was not able to change because it was declared as a *const* variable and thus gave an error

```
// ES5
var name5 = 'Jane Smith';
var age5 = 23;

name5 = 'Jane Miller';
console.log(name5);

age5 = 24;
console.log(age5);

//ES6
const name6 = 'Jane Smith';
let age6 = 23;

age6 = 25;
console.log(age6);

name6 = 'Jane Miller';
console.log(name6)
```

```
Jane Miller
24
25
▸Uncaught TypeError: Assignment to constant variable.
    at script.js:19
```

2. Now let's look at a function example
    - Notice how the 'firstName' was no longer recognized outside of the if block.
    - This is because *let* and *const* are block scoped.
    - **Block** is the code enclosed by the curly brackets

```
// ES5
function driversLicence5(passedTest) {
    if (passedTest) {
        var firstName = 'John';
        var yearOfBirth = 1990;
    }

    console.log(firstName + ', born in ' + yearOfBirth +
    ' is now officially allowed to drive a car.');
}

driversLicence5(true);

//ES6
function driversLicence6(passedTest) {
    if (passedTest) {
        let firstName = 'John';
        const yearOfBirth = 1990;
    }

    console.log(firstName + ', born in ' + yearOfBirth +
    ' is now officially allowed to drive a car.');
}

driversLicence6(true);
```

```
John, born in 1990 is now officially allowed to drive a car.
▸Uncaught ReferenceError: firstName is not defined
    at driversLicence6 (script.js:45)
    at script.js:48
```

- One way of solving this error is to declare the variable outside the if statement just like the following:

```
//ES6
function driversLicence6(passedTest) {
    let firstName;
    const yearOfBirth = 1990;

    if (passedTest) {
        firstName = 'John';
    }

    console.log(firstName + ', born in ' + yearOfBirth +
    ' is now officially allowed to drive a car.');
}

driversLicence6(true);
```

- A use case for this important characteristic is using the same variable name for different blocks such as for loops
  - Notice how the 'I' declared in the outside block is different from the one declared inside the for loop block

```
let i = 23;

for (let i = 0; i < 5; i++) {
    console.log(i);
}

console.log(i);
```

```
0
1
2
3
4
23
```

3. Now lets take a look at example that demonstrate the difference between *let* and *var* declarations when the variable is used before its declaration
   - Is the *let* scenario, we can only use a variable AFTER we have declared it

```
//ES5
console.log(title5);
var title5;

//ES6
console.log(title6);
let title6;
```

```
undefined
▸ Uncaught ReferenceError: Cannot access 'title6' before initialization
    at script.js:57
```

105. Blocks and IIFEs
- Data privacy can now be obtained using blocks, let, and const
- Recall that let and const are block scoped meaning that the variables declared inside the curly brackets{} are not accessed outside of the block where the variables were declared

- For example, the variables will not be accessed outside its block and will throw an error as follow:

```
{
    const a = 1;
    let b = 2;
}

console.log(a + b);
```

```
▸Uncaught ReferenceError: a is not defined
    at script.js:94
```

106. Strings
- We can now use backticks and template literal to avoid the many use of + and ' ' before

```
let firstName = 'John';
let lastName = 'Smith';
const yearOfBirth = 1996;

function calcAge(year) {
    let now = new Date;
    return (now.getFullYear( ) - year);
}

//E55
console.log('This is ' + firstName + ' ' + lastName + '. He was born in ' +
yearOfBirth + '. Today, he is ' + calcAge(yearOfBirth) + ' years old.');

//ES6
console.log(`This is ${firstName} ${lastName}. He was born in ${yearOfBirth}
. Today, he is ${calcAge(yearOfBirth)} years old.`);


const n = `${firstName} ${lastName}`;
console.log(n.startsWith('J'));
console.log(n.startsWith('j'));

console.log(n.endsWith('ith'));
console.log(n.endsWith('smi'));

console.log(n.includes('ohn'));
console.log(n.includes('wassap'));

console.log(n.repeat(5));
console.log(` ${n }`.repeat(5));
```

```
This is John Smith. He was born in 1996. Today, he is 24 years old.
This is John Smith. He was born in 1996. Today, he is 24 years old.
true
false
true
false
true
false
John SmithJohn SmithJohn SmithJohn SmithJohn Smith
 John Smith John Smith John Smith John Smith John Smith
```

107. Arrow Functions: Basic

- Here is how we would have written the *map* method to calculate ages using an array of birth years in ES5

```
const years = [1990, 1965, 1982, 1937];
var now = new Date;
nowYear = now.getFullYear();

//ES5
var ages5 = years.map(function(el) {
    return (nowYear - el);
});
console.log(ages5);
```

- But using the Arrow functions, we can write the above code in a much cleaner way
- There are 3 ways the arrow function can be written
  1. One parameter, one line of code

```
//ES6
let ages6 = years.map(el => nowYear - el);
console.log(ages6);
```

  2. More than one parameter, one line of code

```
// must parenthesis to separate first argument if there are more than one
parameters
ages6 = years.map((el, index) => `Age element ${index + 1}: ${nowYear - el}.
`);
console.log(ages6);
```

  3. More than one parameter, more than one line of code

```
// must use curly brackets if there is more than one line of code
ages6 = years.map((el, index) => {
    age = nowYear - el;
    return `Age element ${index + 1}: ${age}.`
})
console.log(ages6);
```

108. Arrow Functions: Lexical *this* keyword

- **Advantage:** they share surrounding *this* keyword. Arrow function do not have a *this* keyword, and simply use the *this* keyword in the function that they are written in.

- Recall that in ES5, during a regular function call, the *this* variable points to the global context. A workaround to this is by defining a variable as the this keyword *this* and then use this variable in replacement of the *this* keyword. For Example:

```
//ES5
var box5 = {
    color: 'green',
    position: 1,
    clickMe: function() {
        var self = this;
        document.querySelector('.green').addEventListener('click', function
        () {
            var str = 'This box number ' + self.position + ' and it is ' + '
            and it is ' + self.color + '.';
            alert(str);
        });
    }
}

box5.clickMe();
```

- Another workaround to this problem is by using the bind method on the function. Because the bind method copies the function and passes the this variable, it then points again to its outer object. For example:

```
//ES5
Person.prototype.myfriends5 = function(friends) {
    var arr = friends.map(function(el) {
        return this.name + ' is friends with ' + el;
    }.bind(this));
    console.log(arr);
}

var friends = ['Bob', 'Jane', 'Mark'];
new Person('John').myfriends5(friends);
```

- In ES6 however, we can use the power of the arrow function and its advantage.
- The code above can be written using the arrow function without the *this* keyword pointing to the global context (without the work around) as shown below
    - The *this* keyword inside the arrow function is essentially from the function is it written in, thus it point to the box6 object instead of the global object.

```
//ES6
const box6 = {
    color: 'green',
    position: 1,
    clickMe: function() {
        document.querySelector('.green').addEventListener('click', () => {
            var str = 'This box number ' + this.position + ' and it is ' + '
            and it is ' + this.color + '.';
            alert(str);
        });
    }
}

box6.clickMe();
```

- BUT! Be careful with its use as it is easy to lose track of where the *this* keyword points to when there are nested arrow functions

109. Destructuring
- Recall that in ES5 we had to deconstruct an array and assign each element to a variable such as the following:

```
//ES5
var john = ['John', 26];
var name5 = john[0];
var age5 = john[1];
```

- Now imagine if the array was much bigger, and we had to assign each element of a 20+ array to a new variable…this can be quite tedious.
- IN ES6, this can be short-handed by coding in the following way

```
//ES6
const [name6, age6] = ['John', 26];
console.log(name6);
console.log(age6);
```

- More examples of using destructuring (object, function) below:
- ** if data was constructed using [] or {} use the same to deconstruct**

```
const obj = {
    firstName: 'John',
    lastName: 'Smith'
};

const {firstName, lastName} = obj;
console.log(firstName);
console.log(lastName);

// if we want to change the variable name we can do the following
const {firstName: a, lastName: b} = obj;
console.log(a);
console.log(b);


function calcAgeRetirement(year) {
    const age = new Date().getFullYear() - year;
    return [age, 65 - age];
}

const [age2, retirement] = calcAgeRetirement(1990);
console.log(age2);
console.log(retirement)
```

```
John
Smith
John
Smith
30
35
```

110. Arrays

- *From* method
  - Recall that in ES5, sometimes we want to use array methods on lists. But in order to do this, we must do this trick where we convert the list to an array using the call method on the array prototype as follows:

```
//ES5
var boxesArr5 = Array.prototype.slice.call(boxes);
boxesArr5.forEach(function(cur) {
    cur.style.backgroundColor = 'dodgerblue';
});
```

  - But there is now a simpler way of doing this in ES6, and that is with the use of the *from* method. See code snippet below that implements the above code in ES6 version

```
//ES6
const boxesArr6 = Array.from(boxes);
boxesArr6.forEach(cur => cur.style.backgroundColor = 'dodgerblue');
```

- *Const* [element] *of* [array]
  - If we want to use *continue* or *break* in any of the iteration in the loop, we would not be able to use *forEach* in this case. In ES5 it would have to be implemented as follows:

```
//ES5
for (var i = 0; i < boxesArr5.length; i++) {
    if (boxesArr5[i].className === 'box blue') {
        continue; //skips the iteration of this loop and moves on to the next
    }
    boxesArr5[i].textContent = 'I changed to blue!';
}
```

  - However, in ES6, we can iterate through an array similar to *forEach* but also allows the use of *continue* or *break* in the loop using the *const of* operator. An example of how this can be implemented is shown below:

```
//ES6
for (const cur of boxesArr6) {
    if (cur.className.includes('blue')) {
        continue;
    }
    cur.textContent = 'I changed again to blue!';
}
```

- *Find* and *findIndex*
  - Recall that in ES5, to find a specific elements satisfying a certain condition in an array we needed to do the following steps:

```
//ES5
var ages = [12, 17, 8, 21, 14, 11];

var full = ages.map(function(cur) {
    return cur >= 18;
});
console.log(full.indexOf(true));
console.log(ages[full.indexOf(true)]);
```

```
3
21
```

- But in ES6, this can be simplified using the *findIndex* or the *find* method as follows:

```
console.log(ages.findIndex(cur => cur >= 18));
console.log(ages.find(cur => cur >= 18));
```

```
3
21
```

111. <u>The Spread Operator</u>

- Consider the following code below. Let's say that we have an array with data that we want to input as arguments in the *addFourAges()* function where each of the element in the array must be passed as individual arguments.

```
function addFourAges(a, b, c, d) {
    return a + b + c + d;
}

var ages = [18, 30, 12, 21];
```

- To do this in ES5, we can use the apply method like as follows. Recall that the apply method is similar to the call method, but it accepts array as inputs

```
var sum2 = addFourAges.apply(null, ages);
console.log(sum2);
```

```
81
```

- But there is a clearer way of doing this in ES6 and that is by using the spread operator. The spread operator opens up and expands the element of an array.

```
//ES6
const max3 = addFourAges(...ages);
console.log(max3);
```

```
81
```

- It can also be useful in joining arrays together. For example:

```
const familySmith = ['John', 'Jane', 'Mark'];
const familyMiller = ['Mary', 'Bob', 'Ann'];
const bigFamily = [...familySmith, 'Lily', ...familyMiller];
console.log(bigFamily);

const h = document.querySelector('h1');
const boxes = document.querySelectorAll('.box');
const all = [h, ...boxes];
Array.from(all).forEach(cur => cur.style.color = 'purple');
```

```
▶ (7) ["John", "Jane", "Mark", "Lily", "Mary", "Bob", "Ann"]
```

112. Rest Parameters

- The rest parameters looks exactly the same as the spread operators, but are different. They are quite the opposite in fact.
- That is because the rest Parameters takes in single parameters and turns it into an array when we call a function with multiple parameters
- The rest parameter is used during function declaration while the spread operator is used in function calls

- First, let us take a look at a function written in ES5. We want to grab the arguments that were passed as parameters and iterate through them using the *forEach* method.
  - Note that the arguments are *not* structured as an array
  - We must convert it to array first (using the splice trick)

```
//ES5
function isFullAge5() {
    console.log(arguments); //note that arguments not an array
    var argsArr = Array.prototype.slice.call(arguments);
    argsArr.forEach(function(cur) {
        console.log((2020 - cur) >= 18);
    })
}

isFullAge5(1990, 2005, 1965, 2016, 1987);
```

```
▼Arguments(5) [1990, 2005, 1965, 2016, 1987, callee: ƒ, Symbol(Symbol.iterator): ƒ]
   0: 1990
   1: 2005
   2: 1965
   3: 2016
   4: 1987
  ▶callee: ƒ isFullAge5()
   length: 5
  ▶Symbol(Symbol.iterator): ƒ values()
  ▶__proto__: Object
true
false
true
false
true
```

- Now lets take a look at how this can be simplified using the rest parameter in ES6.
    - Notice how the arguments have been automatically converted to an array

```
//ES6
function isFullAge6(...years) {
    console.log(years);
    years.forEach(cur => console.log((2020 - cur) >= 18));
}

isFullAge6(1990, 2005, 1965, 2016, 1987);
```

```
▼(5) [1990, 2005, 1965, 2016, 1987]
   0: 1990
   1: 2005
   2: 1965
   3: 2016
   4: 1987
   length: 5
  ▶__proto__: Array(0)
true
false
true
false
true
```

113. Default Parameters
    - We use default parameters whenever we want one or more parameters of a function to be preset, so we want them to have a default value.

    - Let's first code this in ES5. JavaScript allows function calls to run even if not all parameters were defined. Those parameters would simply be assigned as *undefined*.
        - To achieve preset arguments for parameters, we can use an if statement

```
//ES5
function SmithPerson(firstName, yearOfBirth, lastName, nationanlity) {

    lastName === undefined ? lastName = 'Smith' : lastName = lastName;
    nationanlity === undefined ? nationanlity = 'american' : nationanlity =
    nationanlity;

    this.firstName = firstName;
    this.yearOfBirth = yearOfBirth;
    this.lastName = lastName;
    this.nationanlity = nationanlity;
}

var john = new SmithPerson('John', 1990);
var emily = new SmithPerson('Emily', 1983, 'Diaz', 'Spanish');
```

```
> john
SmithPerson {firstName: "John", yearOfBirth: 1990, lastName: "Smith", nationanlity: "american"}
    firstName: "John"
    lastName: "Smith"
    nationanlity: "american"
    yearOfBirth: 1990
  ▶ __proto__: Object
> emily
SmithPerson {firstName: "Emily", yearOfBirth: 1983, lastName: "Diaz", nationanlity: "Spanish"}
    firstName: "Emily"
    lastName: "Diaz"
    nationanlity: "Spanish"
    yearOfBirth: 1983
  ▶ __proto__: Object
```

- ES6 allows us to do this in a much simpler way without having to write if statements.
  - The default parameters can be set directly in the function parameters.

```
//ES6
function ChadPerson (firstName, yearOfBirth, lastName = 'Smith',
nationanlity = 'american') {
    this.firstName = firstName;
    this.yearOfBirth = yearOfBirth;
    this.lastName = lastName;
    this.nationanlity = nationanlity;
};

var bob = new ChadPerson('Bob', 1999);
var ann = new ChadPerson('Ann', 1967, 'Pierre', 'French');
```

```
> bob
ChadPerson {firstName: "Bob", yearOfBirth: 1999, lastName: "Smith", nationanlity: "american"}
    firstName: "Bob"
    lastName: "Smith"
    nationanlity: "american"
    yearOfBirth: 1999
  ▶ __proto__: Object
> ann
ChadPerson {firstName: "Ann", yearOfBirth: 1967, lastName: "Pierre", nationanlity: "French"}
    firstName: "Ann"
    lastName: "Pierre"
    nationanlity: "French"
    yearOfBirth: 1967
  ▶ __proto__: Object
```

114. <u>Maps</u>
- This is an entirely new data structure in ES6
- Map is an object. You can 'entries' to an instance of this object using the *set* method
- Each 'entry' has a *key* and a *value* (ex: question.set([key], [value]))
- You can also use *get* method to retrieve values using its key
- You can use the *size* property to determine how many entries are in a map object

```
const question = new Map();

question.set('question', 'What is the official name of the latest major
JavaScript version?');
question.set(1, 'ES5');
question.set(2, 'ES6');
question.set(3, 'ES2015');
question.set(4, 'ES7');
question.set('correct', 3);
question.set(true, 'Correct Answer');
question.set(false, 'Wrong answer');

console.log(question.get('question'));
console.log(question.size);

if (question.has(4)) {
    question.delete(4);
}
console.log(question.get('question'));
console.log(question.size);
```

```
What is the official name of the latest major JavaScript version?
8
What is the official name of the latest major JavaScript version?
7
```

- Maps are also iterable, meaning that its entries can be iterated using loops.
- Heres an example using the forEach method

```
question.forEach((value, key) => console.log(`This is ${key}, and it's set
to ${value}`));
```

```
This is question, and it's set to What is the official name of the latest major JavaScript
version?
This is 1, and it's set to ES5
This is 2, and it's set to ES6
This is 3, and it's set to ES2015
This is correct, and it's set to 3
This is true, and it's set to Correct Answer
This is false, and it's set to Wrong answer
```

- Destructuring is also helpful if you want to access both the key and the value of an entry using the *for of* loop.
- What is helpful with maps, is that we can pick out certain values using characteristics of its key (ie. If it's a number for example)

```
for (let [key, value] of question.entries()) {
    if (typeof(key) === 'number') {
        console.log(`Answer ${key}: ${value}`);
    }
}

const ans = parseInt(prompt('Write the correct Answer'));
console.log(question.get(ans === question.get('correct')));
```

```
Answer 1: ES5
Answer 2: ES6
Answer 3: ES2015
Correct Answer
```

- Here are some reasons why maps are better than objects in creating hashmaps
  1. Keys can be anything which opens up many possibilites
  2. Maps are easily iterated through
  3. Really easy to get the size of the map
  4. Easily add or remove data from a map

115. Classes
- Classes does not really add anything new to ES6.
- It is just something we call synthetic sugar as it simply means that classes makes it easier to write objects and implement inheritance

- Let's first take a look at how we implement constructors and instances of this constructor in ES5

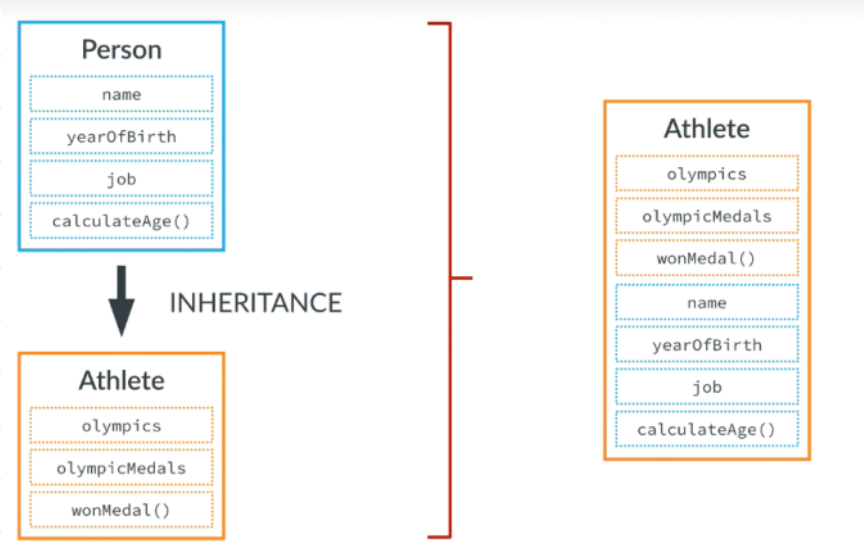
```
//ES5
var Person5 = function(name, yearOfBirth, job) {
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}

Person5.prototype.calculateAge = function() {
    var age = new Date().getFullYear() - this.yearOfBirth;
    console.log(age);
}

var john5 = new Person5('John', 1990, 'teacher');
```

- Now let us now take a look at how this is implemented using classes in ES6
  - The keyword *class* is used
  - The keyword *constructor* is used
  - Methods inherited can be written directly inside the class
  - *Static* methods are methods that are associated with the class itself and not the instance

```
//ES6
class Person6 {

    constructor (name, yearOfBirth, job) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.job = job;
    }

    calculateAge() {
        var age = new Date().getFullYear() - this.yearOfBirth;
        console.log(age);
    }

    static greeting() {
        console.log('Hey there!');
    }
}

const john6 = new Person6('John', 1990, 'teacher');
Person6.greeting();
```

- Two things to note
  1. Class definitions are not hoisted meaning that we must first implement a class, and only later in our code can we start using it
  2. We can only add methods to classes but not properties.

116. <u>Classes with Subclasses</u>



- Below is how the above structure is coded in ES5
  - Note that to establish the prototype chain we must declare Athelete5 to be an object of Person5. This connects the two protope

- Only ONCE the prototype chain has been ESTABLISHED can we create methods specific to the Athlete constructor

```
//ES5
var Person5 = function(name, yearOfBirth, job) {
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}

Person5.prototype.calculateAge = function() {
    var age = new Date().getFullYear() - this.yearOfBirth;
    console.log(age);
}

var Athelete5 = function(name, yearOfBirth, job, olympicGames, medals) {
    Person5.call(this, name, yearOfBirth, job);
    this.olympicGames = olympicGames;
    this.medals = medals;
}

Athelete5.prototype = Object.create(Person5.prototype);

// must be written after the prototype chain has been established
Athelete5.prototype.wonMedal = function() {
    this.medals++;
    console.log(this.medals);
}

var johnAthelete5 = new Athelete5('John', 1990, 'swimmer', 3, 10);

johnAthelete5.calculateAge();
johnAthelete5.wonMedal();
```

- Below is how the above structure is coded in ES6
  - Use *super* method to call the super class

```
//ES6
class Person6 {

    constructor (name, yearOfBirth, job) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.job = job;
    }

    calculateAge() {
        var age = new Date().getFullYear() - this.yearOfBirth;
        console.log(age);
    }
}

class Athelete6 extends Person6 {

    constructor(name, yearOfBirth, job, olympicGames, medals) {
        super(name, yearOfBirth, job);
        this.olympicGames = olympicGames;
        this.medals = medals;
    }

    wonMedal() {
        this.medals++;
        console.log(this.medals);
    }
}

var johnAthelete6 = new Athelete6('John', 1990, 'swimmer', 3, 10);
johnAthelete6.calculateAge();
johnAthelete6.wonMedal();
```

SECTION 8: Asynchronous JavaScript: Promises, Async/Await and Ajax

120. An example of Asynchronous JavaScript
    - **Asynchronous JavaScript**: is the code that runs in the background
    - For example:
        - *setTimeOut()* function accepts a call back function and a time (in ms).

```
const second = () => {
    setTimeout(() => {
        console.log('Async Hey There!');
    }, 2000);
}

const first = () => {
    console.log('Hey There!');
    second();
    console.log('The end');
}
```

```
Hey There!
The end
```

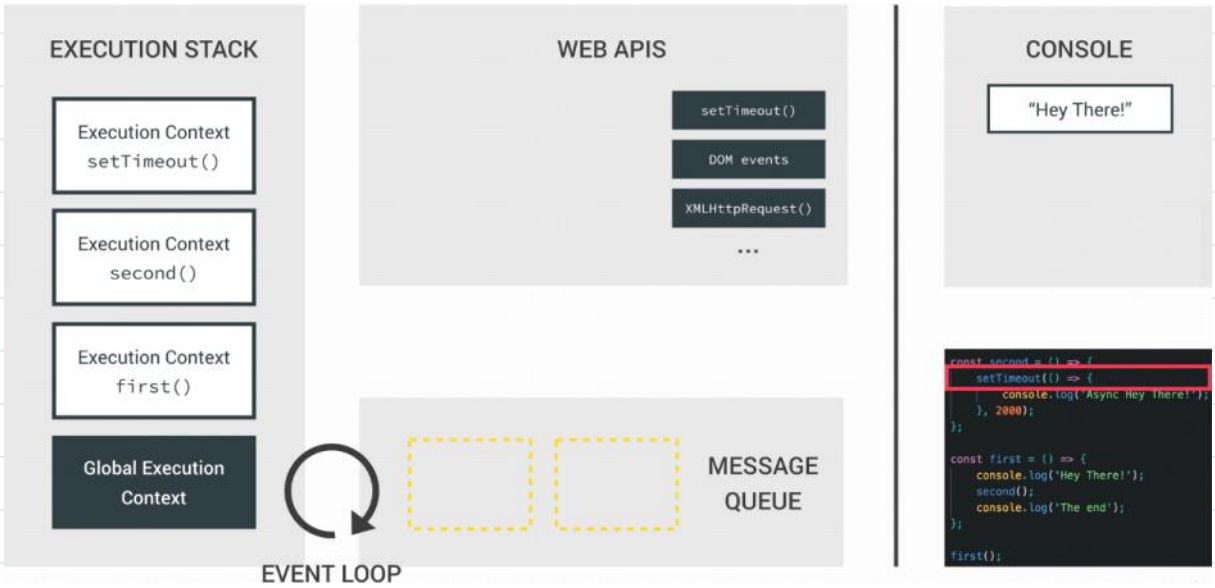
After 2 seconds... ==>

```
Hey There!
The end
Async Hey There!
```

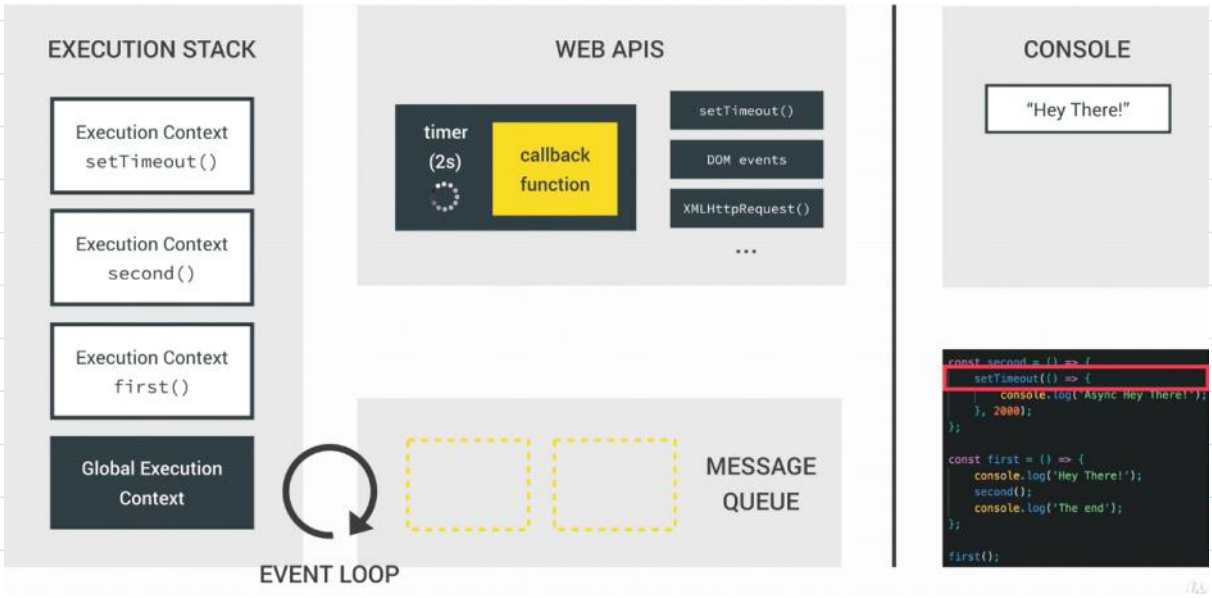
121. Understanding Asynchronous JavaScript: The Event Loop
    - Consider a scenario where we take an image from our DOM and pass it to a image processing function. We know that this function will take some time to process and it becomes a problem when we let the rest of our code have to wait.
    - This is where async JavaScript comes in to play because we do not have to wait for a function to finish its work. Instead we can let it do its job in the background, so that we can move on with our code execution.
    - We can then also pass in a callback function as soon as the main function has done whatever it had to do. We then move on immediately so that the code is never blocked. Meaning that it can keep processing our code line by line.
    - We can use callback functions to defer actions into the future in order to make our code non blocking.

    - **Here is how it works**
        - Event loop as well as web API's together with the execution stack and the message queue make up our JavaScript run time.
        - Note that web APIs and other stuff such as DOM manipulation methods, Set Timeout, HTTP requests for AJAX, geolocation, local storage etc live outside the JavaScript Engine. We just hace access to them because there are also in the JavaScript runtime

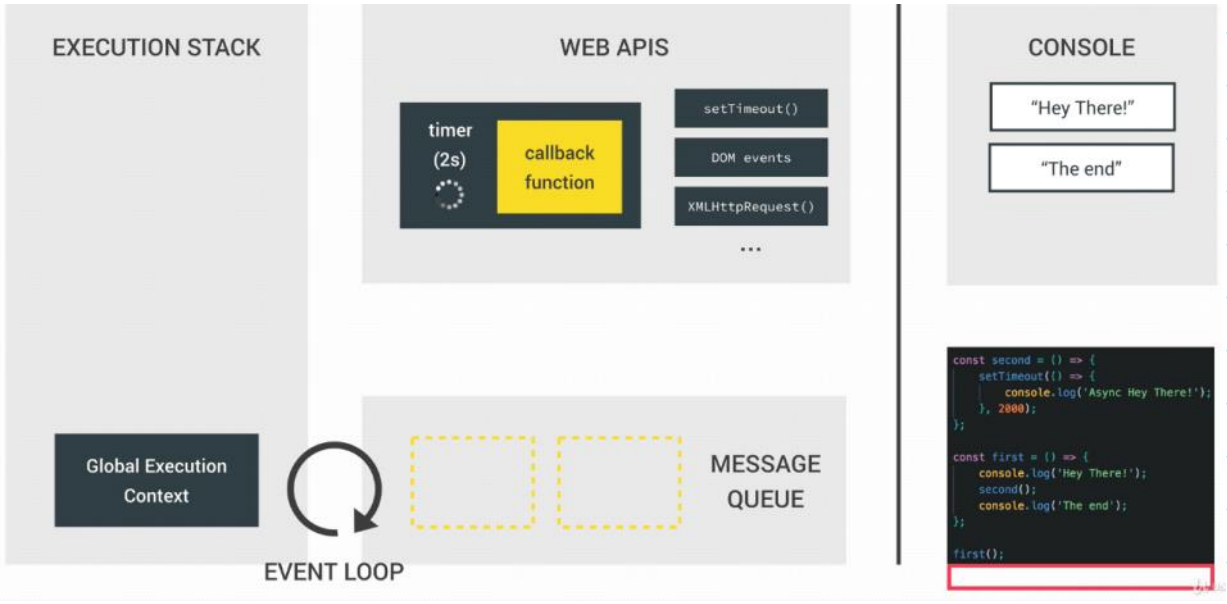
    i. The code runs synchronously and EC gets pushed on to the stack.



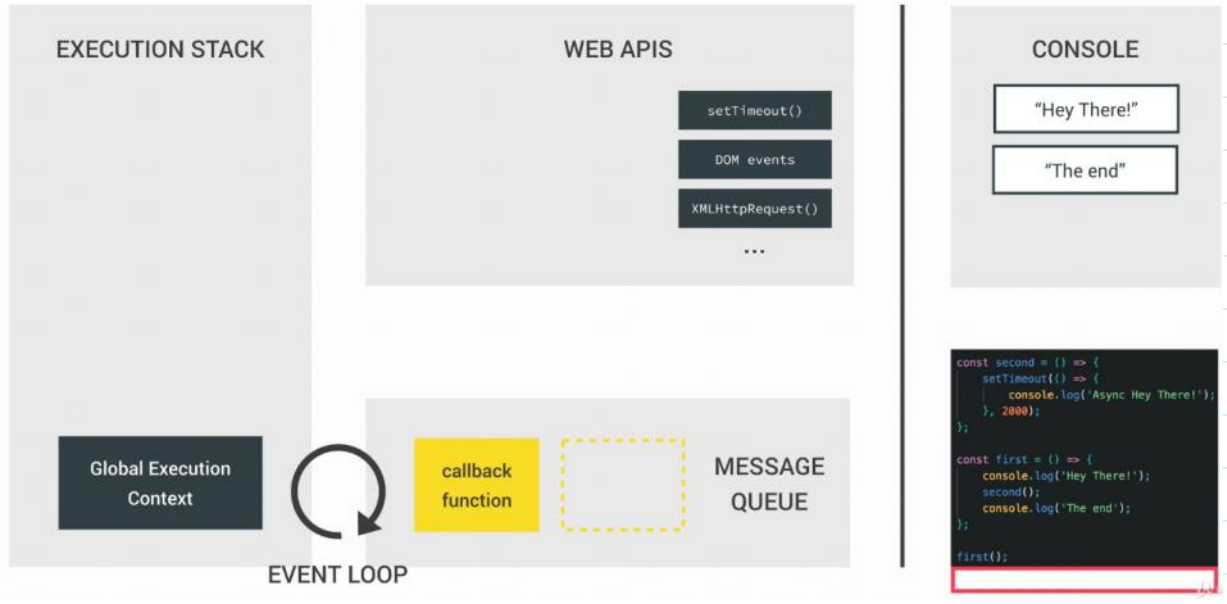
i. When our *Set Timeout* function is called, it is created together with the timer and sits inside the Web API until it finishes its work all in an Asynchronous way. The call back function is not called right now.



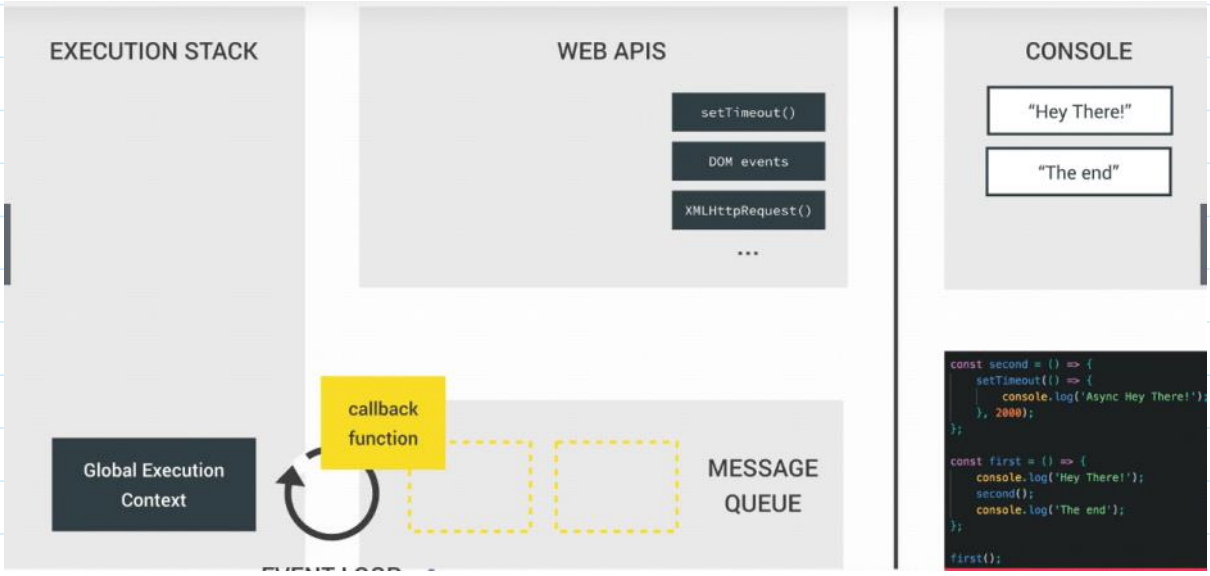
i. The rest of our code runs synchronously again.



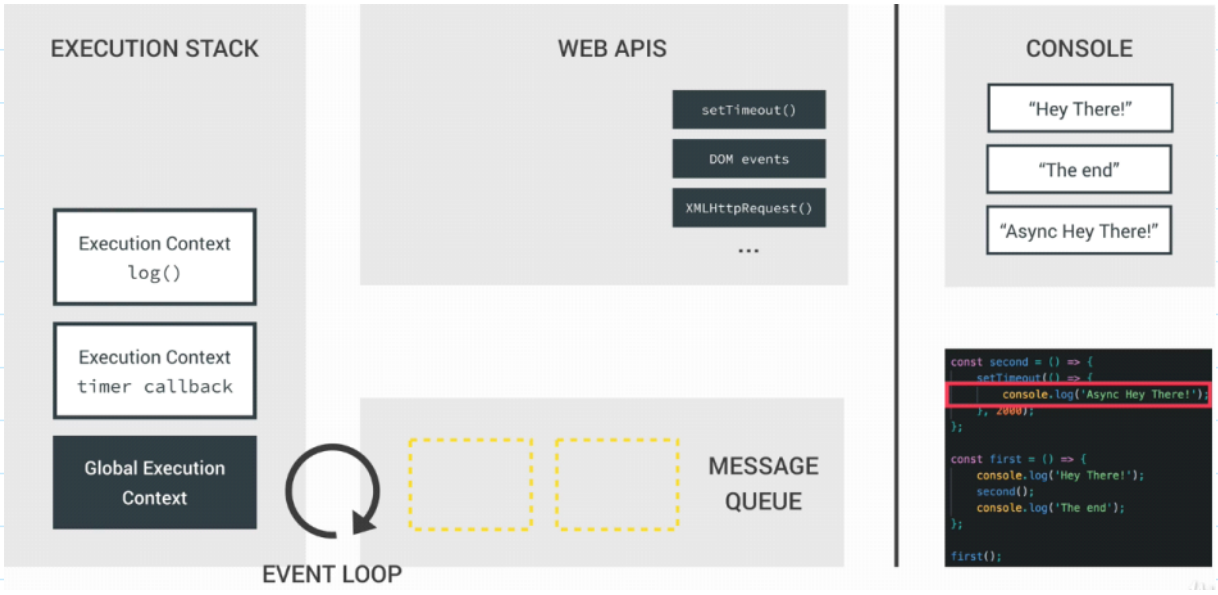
i. After the timer has finished, the callback function is placed on the message queue and waits to be called



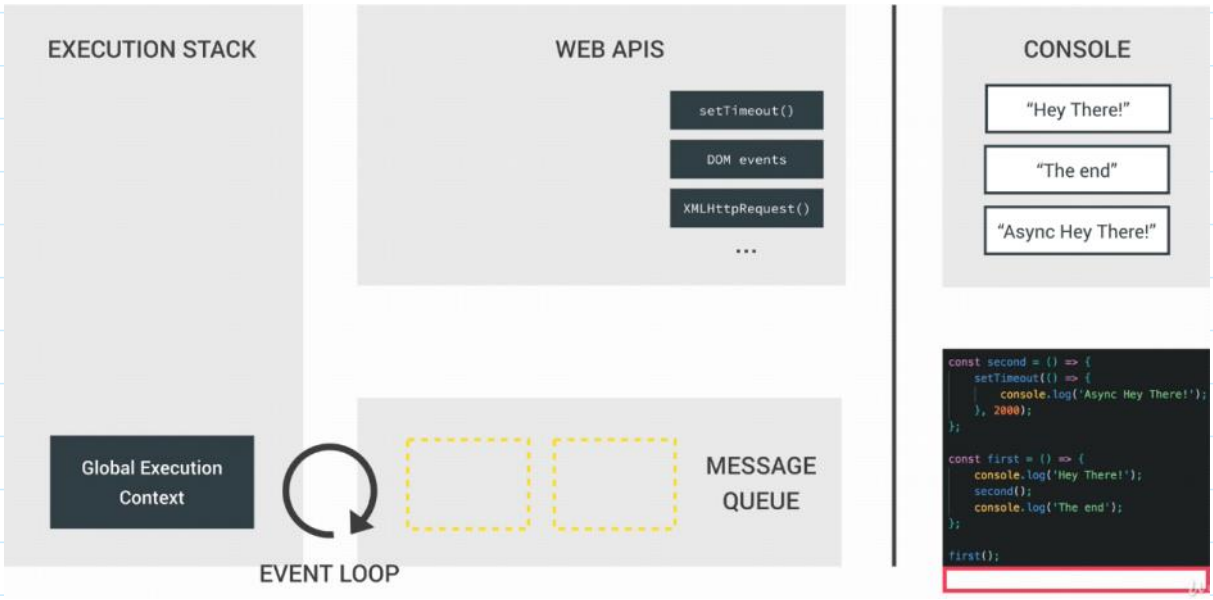
i. The Event loop actively checks the message queue and pushes it to the execution stack once the stack is empty



i. Now, the call back function is called and an EC is placed on the stack. The code inside the callback function then runs.



i. Until it has finished



122. <u>The Old Way: Asynchronous JavaScript with Callbacks</u>
    - In this exercise were going to simulate ajax calls to get some fake recipes using set timeouts.
        - Simulate loading data from a remote web server (asynchronous using set timeout)
    - Below is an example of something called Call Back Hell
        - There is a callback within a call back within a call back.
        - If were dealing with many of these, then it becomes unmanagable
        - To prevent this, JavaScript has something called **promises**

```
function getRecipe() {
    setTimeout(() => { //simulates ID coming from remote server
        const recipeID = [523, 883, 432, 974];
        console.log(recipeID);

        setTimeout((id) => { //simulates the reciep coming from
        remote server
            const recipe = {
                title: 'Fresh Tomato Pasta',
                publisher: 'Jonas'
            }
            console.log(`${id}: ${recipe.title}`);

            setTimeout(publisher => {
                const recipe2 = {
                    title: 'Italian Pizza',
                    publisher: 'Jonas'
                }
                console.log(`${recipe2.title} made by ${recipe2.
                publisher}`);
            },1500), recipe.publisher;

        },1500, recipeID[2]);

    }, 1500);
}
getRecipe();
```

123. From Call Back Hell to Promises

- **Promise**:
  - Object that keeps track whether an asynchronous event has happened already or not.
  - Determines what happens after the event has happened.
  - Implements the concept of a future value that we are expecting
  - 'Promises to do something in the background'

- Promise States:
  - Pending: "Promise is pending before the event has happened"
  - Called/Settled/Resolved: "Promise is called after the event has happened"
  - Fulfilled: "Promise is fulfilled if it produces a result"
  - Rejected: "There was an error"

- In practical terms, we can 'produce' and 'consume' promises
  - When we produce a promise, we create a promise and send a result using that promise
  - When we consume a promise, we can use callback functions for fulfillment or for rejection of our promise

- The function passed on to the promise function is called the 'executor function'
  - The executor function takes in two parameters which are *resolve*, and *reject*
  - It is used to inform the promise whether the event handling was successful or not
  - If successful, we call the resolve function, is not successful, we call the reject function

```
const getIDs = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve([523, 883, 432, 974]);
        // setTimeout will aylways be succesful no need for a reject
        reject('error');
    }, 1500);
});
```

- We can use two methods to consume a promise: *then* and *catch* methods
  - All promise objects inherit these two methods
  - These methods has a callback function witch can take a an argument.
  - The then method takes in parameters passed on to the resolve function.
  - The catch method takes in parameters passed on to the reject function.

```
getIDs.then(IDs => {
    console.log(IDs);
})
.catch(error => {
    console.log(error)
});
```

- Here is the implementation of the above call back hell example using promises
  - There are three promises meaning that there will be three .then methods and one .catch method when something fails.
  - For each promise there is one resolve for when the event is successful. The parameters of this resolve method is passed on to the callback function parameters of the .then method.
  - The order of .then methods is a chain that matches the chain of the promises. The first .then method corresponds to the resolve of the first promise, the second .then method corresponds to the resolve of the second promise and so on and so forth.

```
const getIDs = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve([523, 883, 432, 974]);
        // setTimeout will aylways be succesful no need for a reject
        //reject('error');
    }, 1500);
});

const getRecipe = recID => {
    return new Promise((resolve, reject) => {
        setTimeout(ID => {
            const recipe = {
                title: 'Fresh Tomato Pasta',
                publisher: 'Jonas'
            };
            resolve(`${ID}: ${recipe.title}`);
        }, 1500, recID);
    });
};

const getRelated = publisher => {
    return new Promise((resolve, reject) => {
        setTimeout(pub => {
            const recipe2 = {
                title: 'Italian Pizza',
                publisher: 'Jonas'
            };
            resolve(`${recipe2.title} made by ${pub}`);
        }, 1500, publisher);
    });
};
```

```
getIDs
.then(IDs => {
    console.log(IDs);
    return getRecipe(IDs[2]);
})
.then(recipe => {
    console.log(recipe);
    return getRelated('Jonas');
})
.then(recipe2 => {
    console.log(recipe2);
})
.catch(error => {
    console.log(error)
});
```

123. <u>From Promises to Async/Await</u>
   - Makes it easier to consume promises (then and catch methods)

   - Asynchronous function is a function that keeps running in the background.
     - This async function returns a promise
     - It can contain more than one await expression
   - Below is an example of *async/await* in practice (with the same promises as above)

- Our promise *getID* has a resolve of an array of IDs, this is stored in the IDs const variable
- Next we wait until our *getRecipe* promise is resolved and its *resolve* is stored in the recipe variable. Don't forget that this specific promise requires an argument.
- Same thing happens with the *getRelated* promise

```
async function getRecipesAW() {
    const IDs = await getIDs;
    console.log(IDs);
    const recipe = await getRecipe(IDs[2]);
    console.log(recipe);
    const related = await getRelated('Jonas');
    console.log(related);

    return recipe;
}
getRecipesAW().then(result => console.log(`${result} is the best
ever!`));
```

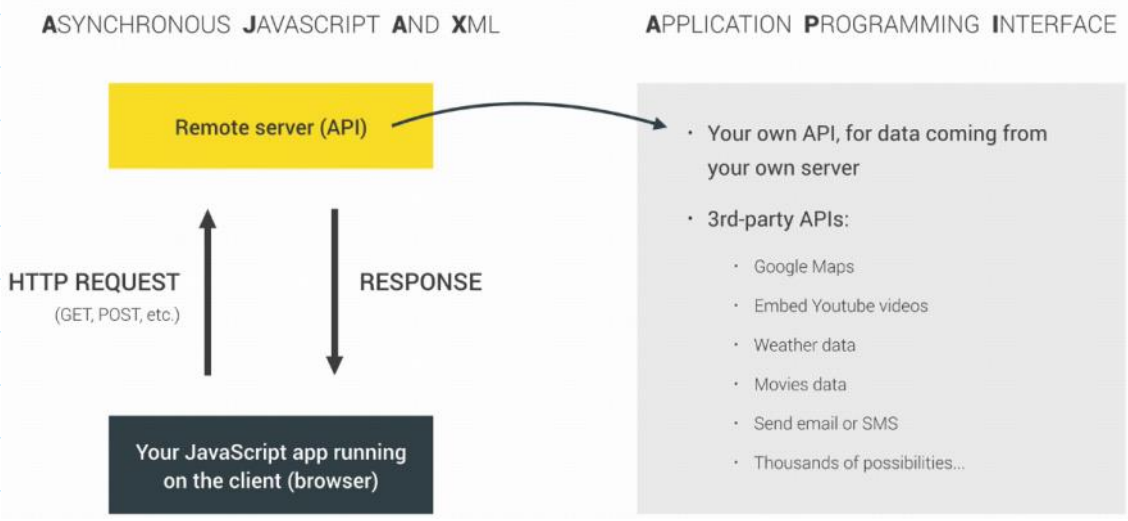
```
▶ (4) [523, 883, 432, 974]
432: Fresh Tomato Pasta
Italian Pizza made by Jonas
432: Fresh Tomato Pasta is the best ever!
```

- Note: the await expression only works within an async function.
- Stopping the async function is not a problem because it does not bother the main function

- Let's say we want to return something from the async function.
  - We cannot simply call the function because once its called, it runs in the background and nothing happens
  - What we must do is use the *then* method because it basically returns a promise with the return value as the resolved value that can be consumed

125. AJAX, and APIs

- **AJAX:** Asynchronous JavaScript And XML
  - It allows us to asynchronously communicate with remote servers
  - For example: we want have our app running in the browser (client) and we want to get some data from our servers but without having to reload the entire page.
  - With AJAX, we can do a simple get HTTP request to our server which will then send back a response containing the data we requested
  - This all happens asynchronously in the background
- **APIs:** Application Programming Interface
  - In a high level perspective its basically a software that can be used by another piece of software in order to allow applications to talk to each other
  - In web dev and AJAX perspective, API is not a server, but is part of a server. It is kind of
  - like an application that receives requests and sends back responses.



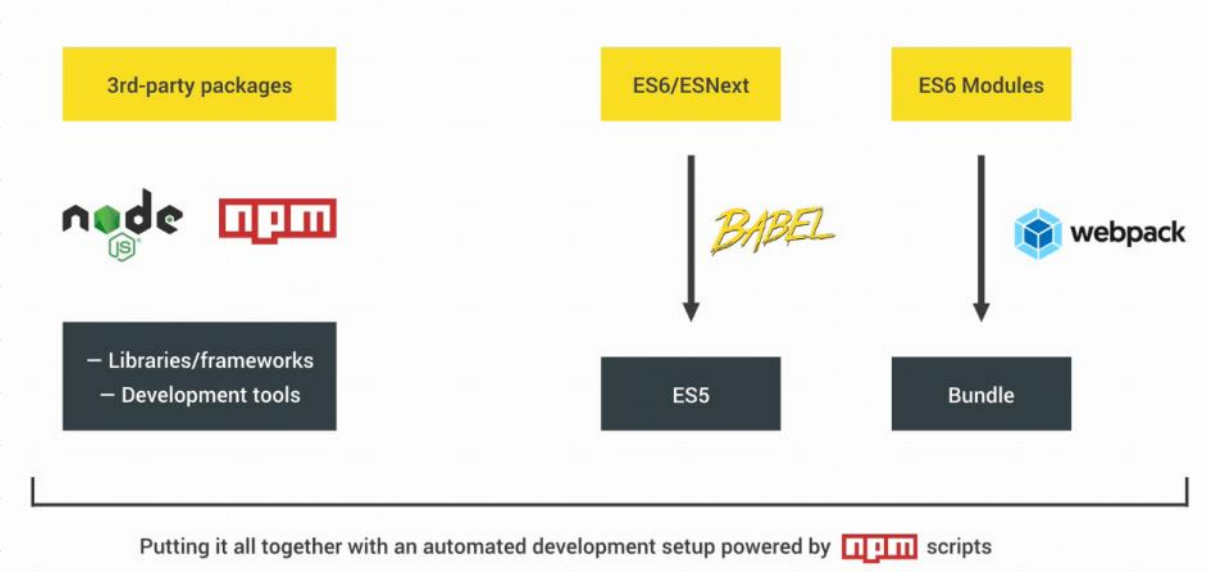
126. Making AJAX calls with Fetch and Promises

- Below is an example of incorporating fetch functions with AJAX calls and APIs
  - Note that fetch function returns a promise which we can then consume
  - URL can be used to fetch the API from a server
    - Sometimes due to different domains, we are not able to access the API
    - To remedy this, we can use something called CORS
    - In this example the string 'https://cors-anywhere.herokuapp.com' was appended to the API URL

```
async function getWeatherAW(woeid) {
    try {
        const result = await fetch(`https://
        cors-anywhere.herokuapp.com/https://www.
        metaweather.com/api/location/${woeid}/`);
        const data2 = await result.json();
        const tomorrow = data2.
        consolidated_weather[1];
        console.log(`Temperatures tomorrow in $
        {data2.title} stay between ${tomorrow.
        min_temp} and ${tomorrow.max_temp}.`);

        return data2;
    }
    catch (error) {
        console.log(error);
    }
}

//getWeatherAW(2487956);

let dataLondon;
getWeatherAW(44418).then(data => {
    dataLondon = data
    console.log(dataLondon);
});
```

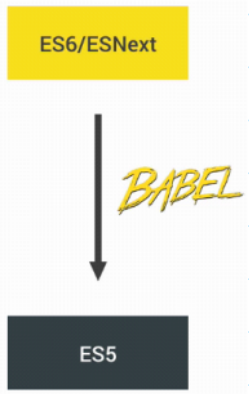
SECTION 9: Modern JavaScript Using ES6, NPM, Babel and Webpack

130. An Overview of Modern JavaScript
   - Up until this point we have coded JavaScript in an old school way
   - Now, the modern JavaScript is not about the language itself but the environment that we use to write in
   - We write JavaScript together with a set of tools that make it easier and better to work with
      - Foundation of these tools: **Node Js** and **NPM** ecosystem
      - This is where we can find third party open source tools, libraries and frameworks needed for modern web development. React, Angular, Jqueary, automtion



   - NPM:
      - Simple command line interface tool that allows us to install and manage these packages
      - It also allows us to write scripts to use for our development tools

- Babel:
  - Converts newer versions of JavaScript like ES6, ES7, ES8 down to ES5
  - So that all browsers are capable of reading the script and running them



- Webpack
  - Bundles modules, code splitting, optimizes

132. <u>A Modern Setup: Installing Node.Js and NPM</u>

1. Install node.js

- Check version in command line
  - ➢ Node -v

2. Create json package for our project. Go to project directory in terminal
   - ➢ Npm init

- This creates a json package right in the project folder containing information about the project (name, publisher, dependencies, description, main, version, etc)

3. Install webpack tool. Go to project directory in terminal
   - ➢ Npm install webpack --save-dev

- The --save-dev option will save the webpack as a developer dependency. Can be checked in

the package.json to be in "devDependencies"

4. Install Jquery library
   - Npm install jquery --save

- This will save it as a dependency only. Can be checked in the package.json to be in "dependencies"

5. To download all the dependencies and modules from another project. Go to project directory in terminal
   - Npm install

- This will read the package and download all the dependencies needed

6. To uninstall a dependency, go to project directory in terminal
   - Npm unistall jquery --save

- Can check json package for successful deletion. It should no longer be in the "dependencies"

7. When installing dependencies using the --save option it saves it locally. Only accessible by that project. To download a dependency and make it accessible globally
   - Npm install live-server --global

- Error might occur if you do not have permission (mac/linux use *sudo* command. Windows use power shell)

134. <u>A Modern Setup: Configuring Webpack</u>

- How to create a basic webpack configuration
- Webpack bundles project into one js, css, file

- Four concept in webpack
  - Entry point : where it will start looking for all the dependencies which it should then bundle together
  - Output : tells webpack where to save our bundle file. Here we pass in an object which contains the <u>absolute</u> path to the folder and then the file name
  - Loaders: allows us to load/import files and more importantly to process them (converting SASS to CSS, ES6 to ES5 JavaScript)
  - Plug ins: allows us to do complex processing of our input files. It receives an array of plugins

1. Create webpack.config.js

   Webpack.config.js

```
const path = require('path');

module.exports = {
    entry: "./src/js/index",
    output: {
        path: path.resolve(__dirname, "dist"),
        filename: "js/bundle.js"
    },
    devServer: {
        contentBase: "./dist"
    }
};
```

2. Add script "dev" and "build" to package.json that will run webpack.config.js
    - The development mode reduces tree shaking and other optimization and compression code usually done during production mode. This speeds up the bundling process

    Package.json

```
"scripts": {
  "dev": "webpack --mode development",
  "build": "webpack --mode production",
```

3. In the command line, run
    - Npm run [script name] (ie: npm run dev) or (ie: npm run build)
    - The bundle.js file should appear inside the ./dist/js folder

135. <u>A Modern Setup: The Webpack Dev Server</u>
- Automatically reloads our page whenever we save our code
- Also provide dev server which will automatically bundle all our JavaScript File and then reloads the ap in the browser whenever we changed a file

1. Add devServer property In webpack.config.js
    - contentBase: specify folder in which webpack should serve our files

    Webpack.config.js

```
const path = require('path');

module.exports = {
    entry: "./src/js/index",
    output: {
        path: path.resolve(__dirname, "dist"),
        filename: "js/bundle.js"
    },
    devServer: {
        contentBase: "./dist"
    }
};
```

2. Add to script in json package. This script will run in the background and updates the browser as soon as we change our code.
- *note that the dev server will take a look at the output path specified in the webpack.config file to know where the wepback will inject the changes to the html file. That path to the html file must be in the output path*

Package.json

```
"scripts": {
  "dev": "webpack --mode development",
  "build": "webpack --mode production",
  "start": "webpack-dev-server --mode development
  --open"
},
```

- Our complete and final code will be in the dist folder. Ready for 'distribution' / production
- Src folder is for development purpose. All our source code goes here. When compiled the webpack bundles everything and into the dist folder in the bundle.js

- We also want to copy our source html into the distribution folder and include the script to our JavaScript bundle when we are making changes to it.

3. We use a plug in for this: install html-webpack-plugin.
   - Npm installhtml-webpack-plugin --save-dev

4. Include the plugin property (which again accepts array) in our webpack config file

Webpack.config.js

```
plugins: [
    new HtmlWebpackPlugin({
        filename: "index.html",
        template: "./src/index.html"
    })
]
```

5. Then re-run the following command
   - Npm run start

136. <u>A Modern Setup: Babel</u>
- Babel is a JavaScript compiler to help us use next gen JavaScript
- Here, we will set up the most straightforward way Babel

1. Install Babel (multiple files)

- Npm install babel-core babel-preset-env babel-loader --save-dev

2. Modify webpack config file to include our loaders
   - Use module property to pass in an object.
   - In there, specify rules property
   - Rules then receives array of all the loaders that we want to use
   - For each loader, we need an object. Each loader need a test property. We will use something called a regular expression for this.
   - We can add the exclude property. We will use this on the node_modules. Because if we wouldn't do this, then Babel would apply to all of the thousands of JavaScript files
   - Then finally use the use property where we can add the loader
   - Overall: this rule tests for .js at the end of the file which means that it is a JavaScript file. It will not scan through the node_modules folder. To all the JavaScript files it finds, it will apply the babel loader

Webpack.config.js

```
module: {
    rules: [
        {
            test: /\.js$/,
            exclude: /node_modules/,
            use: {
                loader: "babel-loader"
            }
        }
    ]
}
```

3. Create a config file for babel
   - Filename: .babelrc
   - *Preset*: is a collection of code transform plug-ins, which are like the pieces of code that actually apply the actual transformations to our code
     - The "env" which stands for environment came from the plugin babel-preset-env
     - Environment, meaning browsers.
   - We want our code to work in the 'last 5 versions' or 'internet explorer less than 8' of browsers then babel automatically figures it out

.babelrc
(modified from lecture to match with babel version)

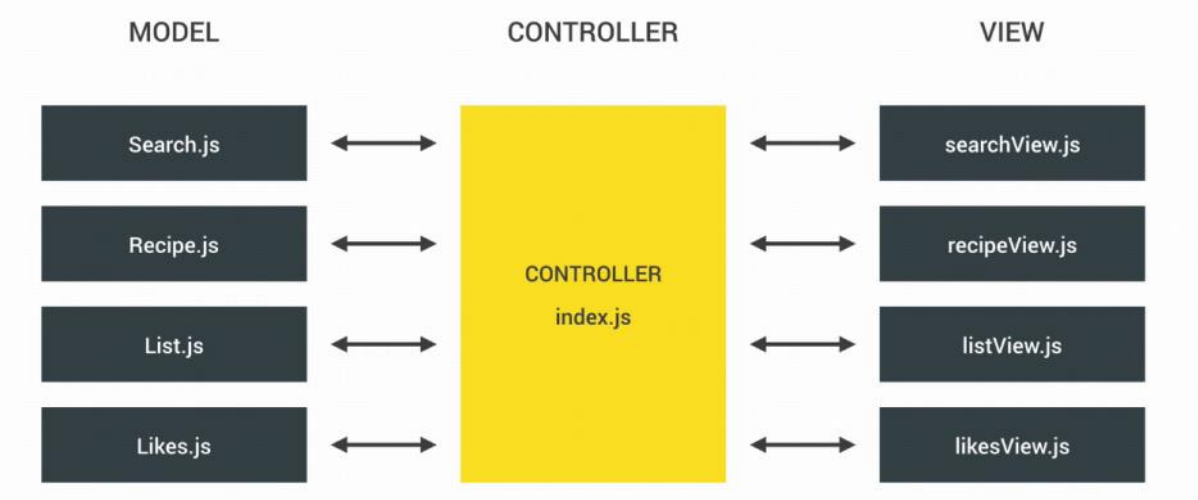
- Right now ES6, ES7, ES8 or later versions of JavaScript will be converted back to ES5
- However, there are some things that we cannot really convert because they simply were not present in the ES5 version language (ie theres not literal english translation for the some words in other languages)
- IE: promises, methods like array.form
- For these we use **Pollyfill**: which basically ES5 code that implements the functionality of the non-convertible ES6 codes (ie: promises)

4. Install babel polyfill
   - npm install babel-polyfill --save
- Again, this is not really a tool, but actual code that will sit inside our project, and so this will be regular dependency and not a dev dependency

5. Add into webpack config
- Add into entry point. With multiple entry point, we now pass an array to the entry property
- Webpack will automatically figure out where the polyfil code is located and will bundle it together with our main code index.js.
6. Installed dependencies to accommodate for newer babel model
   - npm i --save core-js regenerator-runtime
   - npm install --save-dev @babel/preset-env

137. <u>Planning our project architecture with MVC</u>
- MVC: Model-View-Controller Architecture
  - Model: data and logic
  - View: gets and displays data to and from the UI
  - Controller: middle man in between the model and the view
- **Advatange:** nicely decouples the logic from the application logic with a controller in between them that controls the entire apps.
- We will implement this with ES6 JavaScript

- Controller
  - Will include our index.js file. Thanks to ES6, we can make these more modular by separating the different aspects of the app into different files
  - We will have one model and one view for each of the different aspect of the app.
    - EX: aspect -> searching;
    - model -> search.js: where we will do ajax calls
    - view -> searchView.js: where we obtain input string and output resukt
  - The controller brings the model and view together in such a way that the model and view never have to communicate with each other



138. How do ES6 Modules Work
- Using both default and named exports and imports to import and export stuff from and to ES6 modules
- Convention: capitalized for Modules

- Default Export:
  - When we want to export one thing from a module
  - There is no const variable declared

Search.js

```
export default 'I am an exported string';
```

Index.js

```
import str from './models/Search';
```

- Named Export:
  - When we want to export multiple things from a module
  - There are const variable declared to name the things we want to export
  - We call on these names when importing

SearchView.js

```
export const add = (a,b) => a + b;
export const multiply = (a,b) => a * b;
export const ID = 23;
```

Index.js

```
import { add, multiply, ID } from './views/SearchView';
```

- We can also import them and use a different name

Index.js

```
import { add as a, multiply as m, ID } from './views/SearchView';

console.log(`Using imported functions! ${a(ID, 2)} and ${m(3, 5)}. ${str}`)
;
```

- We can also import everything from a module

Index.js

```
import * as searchView from './views/SearchView';
console.log(`Using imported functions! ${searchView.add(searchView.ID, 2)}
and ${searchView.multiply(3, 5)}. ${str}`);
```

140. <u>Making Our First API Call</u>

- Instead of fetch we will use a very popular HTTP request library called axios

1. Install axios
   - This is a code that we want in our project and not a tool
   - ➢ Npm install axios --save

2. Import to our code
   - Recall that with using fetch had a two-step process where we had to convert to json after getting the api
   - But with axion it does the two step process altogether
   - Much better with error handling as well

Index.js

```
import axios from 'axios';

async function getResults(query) {
    try {
        const res = await axios(`https://forkify-api.herokuapp.com/api/
        search?q=${query}`);
        const recipes = res.data.recipes;
        console.log(recipes);
    }
    catch (error) {
        alert(error);
    }
}

getResults('pizza');
```

141. Building the Search Model
    - We will now transfer the code in our index.js file into our Search.js file
    - And change up a few syntax
    - (it was just in the index.js for testing purposes)

142. Building the search controller
    - States
        - What is the current state of our program? Is there something in the shopping cart? Has a recipe been picked? Etc
        - We want this information stored in a variable
            - Search Object
            - Current recipe object
            - Shopping list object
            - Liked recipes

        - We begin with the state being empty and will later be filled as the app is being used and will be empty again when the page is reloaded.

```
const state = {};

const controlSearch = async () => {
    // 1. Get query from view
    const query = 'pizza'; //TODO

    if (query) {
        // 2. New Search object and add to state
        state.search = new Search(query);

        // 3. Prepare UI for results

        // 4. Search for Recipes. getResults returns a promise
        await state.search.getResults();

        // 5. render results on UI only to hapoen after we receive recipes
        console.log(state.search.result);
    }

}
```

143. Building the Search View - Part 1
    - In this lecture we will
        - Learn about advanced manipulation techniques
        - Use ES6 template strings to render entire HTML components
        - How to create a loading spinner
    - Our SearchView.js module will include a bunch of function which we will export, and so we will use named exports and import everything into our controller module (index.js)

1. Create a module for all our DOM strings
    - Will be called Base.js inside views folder
    - Import to global controller (index.js)
2. Inside SearchView.js implement the functions
    - getInput(): return the input value of the field from the DOM
    - renderResults(): iterates through each item in the array and passes it to renderRecipe()
    - renderRecipe(): modifes HTML text using dynamic data inputs and outputs recipe list in DOM with an image, title, and publisher
3. Update controlSearch()

```
// 5. render results on UI only to hapoen after we receive recipes
searchView.renderResults(state.search.result);
```