FRONTEND WEB DEVELOPMENT PROGRAM

Lecture 4: JS Advanced

BeaconFire

OUTLINE

- "use strict"
- Scopes, Hoisting
- Variable Declaration (var, let, const)
- Execution Context, Call Stack, Scope Chain, Lexical Scope
- Closure, Currying, IIFE
- ES6 New Features
- Object-Oriented Programming (Prototypes vs Classes)
- "this", call(), apply(), bind()

BeaconFire

USE STRICT

- **“use strict”**: a directive or literal expression that makes JS execute in strict mode, mostly for writing secure code.
  - Used at the beginning of the script for globally scoped strict mode

- Some effects
  - cannot use undeclared variables, objects, or functions
  - cannot delete variables or objects
  - cannot have duplicate function argument names

- https://www.w3schools.com/js/js_strict.asp

SCOPE

- **Scope**: the section of code that can access a variable or function
  - **Block**: limited to within a non-function body (the {} in if, for, while, …), introduced in ES6
  - **Function**: limited to within a function body (the {})
  - **Global**: all code
    - **window**: a global object that represents the current browser tab
    - https://developer.mozilla.org/en-US/docs/Glossary/Global_object

```
console.log('----block scope----')
if(true){
    //block scope
}

for(var m=0;m<10;m++){
    //block scope
}

{
    //block scope
}

console.log('----function scope----')
function functionScope(){
    //function scope
}

console.log('----global scope----')
var globalV = 'globalV' //global scope
console.log(window.globalV)
```

HOISTING

- **Hoisting**: the code is scanned and all function and variable declarations (not initialization) are done first before executing any code.
  - You can write code that uses a function or variable before actually declaring it.
  - "Function declaration" only applies to the keyword function declaration since the function expression & arrow function are assigned to variables.

DECLARING VARIABLES

Suggestion:
Always use const unless your variable will be reassigned in the future, in which case use let.

	var	(ES6) let	(ES6) const
Hoisted?	Yes initialized to undefined	Yes not initialized, reference error	Yes not initialized, reference error
Scope	Global	Block	Block
Can redeclare?	Yes	No	No
Needs initial value?	No	No	Yes
Can reassign?	Yes	Yes	No
Reference type values: Can mutate properties?	Yes	Yes	Yes

EXECUTION CONTEXT

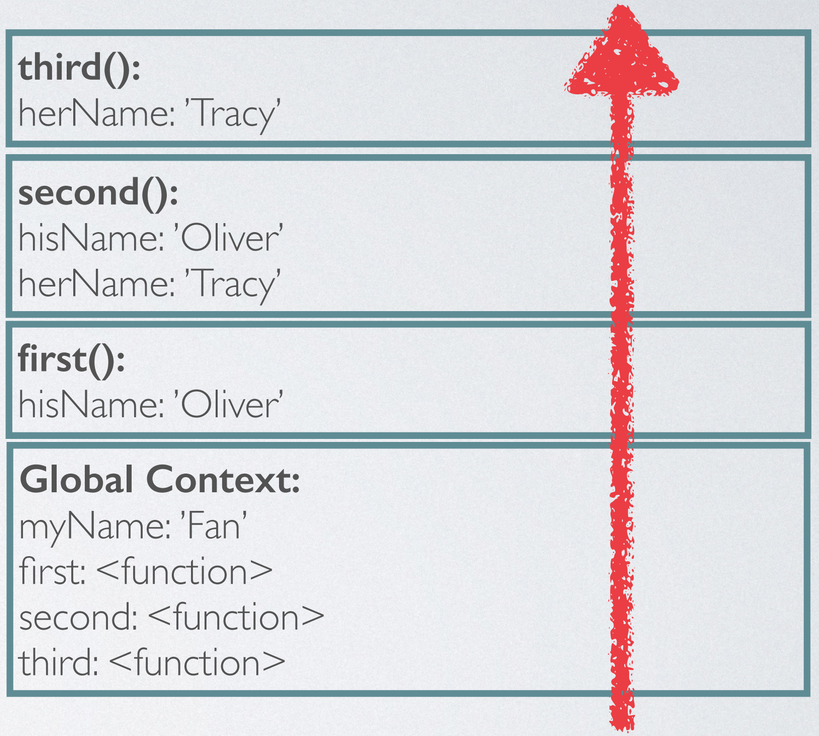
- **Execution context (EC)**: a container for all the information needed to execute some code.
  - By default, one global EC for the top-level code
  - New EC for each function invocation

- EC Phases
  - **Creation**: Store all necessary references to variables and functions in memory.
  - **Execution**: Assign values to variables and invoke functions.

CALL STACK

- **Call stack**: a stack (last-in-first-out, LIFO) that stores all EC created when code is executed
  - For each function invocation, push its EC to the top of the stack

```
var myName = 'Fan'
function first(){
    console.log(`first: my name is ${myName}`)
    let hisName = 'Oliver'
    second(hisName)
    console.log(`first ends`)
}
function second(hisName){
    console.log(`second: his name is ${hisName}`)
    let herName = 'Tracy'
    third(herName)
    console.log(`second ends`)
}
function third(herName){
    console.log(`third: her name is ${herName}`)
    console.log(`third ends`)
}
first()
```



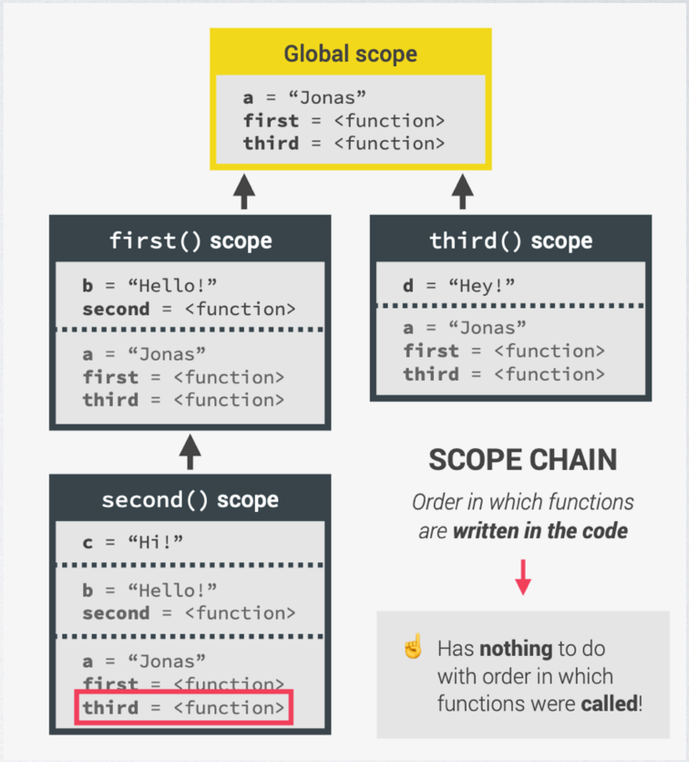
SCOPE CHAIN

- **Scope chain**: the cycle of looking for a variable's value within the current scope, then outer scope, and so on until finding the value or reaching the global scope

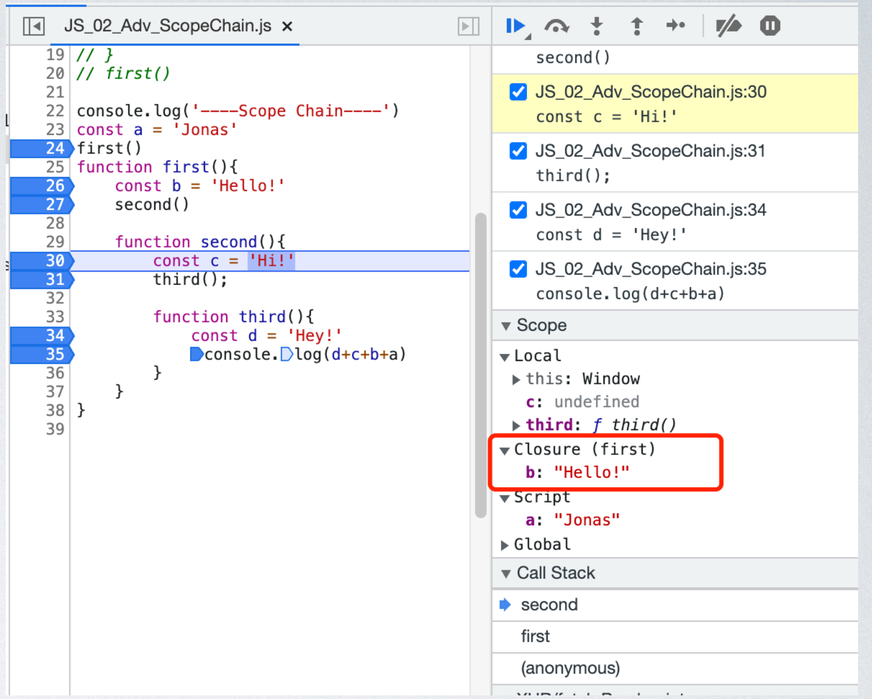
- **Lexical scoping**: code inside a function may access variables defined outside, but not the reverse

```
const a = 'Jonas';
first();
function first() {
  const b = 'Hello!';
  second();

  function second() {
    const c = 'Hi!';
    third();
  }
}
function third() {
  const d = 'Hey!';
  console.log(d + c + b + a);
}
```



CLOSURE



- **Closure**: an inner function with references to outer function scope and its variables (stored in its EC)
  - Use it to mimic private properties or methods in a class, for currying, or callback functions

- "Show your understanding of the language"
  - https://blog.bitsrc.io/closures-in-javascript-why-do-we-need-them-2097f5317daf

- Irrelevant for OOP (classes & private variables), but JS is multi-paradigm (closures are very common in functional programming).

```
function counter() {
  let counter = 0;
  return function () {
    counter++;
    return counter;
  };
}

const add = counter();
console.log(add());
console.log(add());
console.log(add());
```

BeaconFire

CURRYING

- **Currying**: break down a function that takes multiple arguments into a sequence of functions that each take a single argument.
  - Each nested function is a closure.
  - Function composition: functions created by functions

```
const addFn = function (a, b, c) {
  return a + b + c;
};

const addFnCurrying = function (a) {
  return function (b) {
    return function (c) {
      return a + b + c;
    };
  };
};

const arrowAddFnCurrying = a => b => c => a + b + c;
```

- Use-case: refer to closure article in previous slide

IIFE

- **Immediately-Invoked Function Expression (IIFE)**: a function that is invoked when it is defined.
  - Define a function, wrap it in parentheses, invoke it

```
(function hello() { console.log("hello"); })();
(function () { console.log("hello"); })();
(() => { console.log("hello"); })();
```

- Use-cases
  - Avoid polluting the global object by creating a function with its own scope (irrelevant now with let/const)
  - Do some "initiation code" on the top-level once, and clean up variables
  - https://developer.mozilla.org/en-US/docs/Glossary/IIFE

ES6 NEW FEATURES

- What we've covered:
  - Let, Const (block scope)
  - Arrow Functions
  - Template string literals (` `)
  - Map & Set

- Next
  - Array & Object Destructuring
  - Spread & Rest Operator (…)
  - Enhanced object literals
  - Default function parameters
  - Generators
  - Classes
  - Promises, Async/Await (next lecture)
  - Modules (next lecture)

DESTRUCTURING

- **Array Destructuring**: extract values from an array based on the order

```
// ES5: Store elements in variables
const arr1 = [1, 3, 5];
const n1 = arr1[0];
const n2 = arr1[1];
const n3 = arr1[2];
console.log(n1, n2, n3);

// ES6
const [num1, num2, num3] = arr1;
console.log(num1, num2, num3);
```

- **Object Destructuring**: extract values from an object based on matching keys
  - Can destructure nested objects

```
const user = {
  firstName: 'Name',
  say: function () {
    return `hi`;
  },
  devices: {
    phone: 'iphone',
    laptop: 'macbook pro'
  }

};

const { firstName, say: greetings, devices, devices: { laptop } } = user;
```

SPREAD & REST OPERATOR

- **Spread operator (…iterable/enumerable)**: expand an array into its elements, a string into its characters, or an object into its key-value pairs.
  - Pass …arr to a function if it needs multiple arguments
  - Use […arr] or {…obj} to make a shallow-copy.

- **Rest operator (…arr)**: collects any number of arguments and stores them in an array
  - Used in the function declaration to accept "the rest" of the arguments or in destructuring to store "the rest"

```
// Add elements
const arr3 = ['a', 'b', 'c'];
const arr4 = [0, ...arr3, 10];
console.log("arr4 =", arr4);

// Shallow copy
const arr3Copy = [...arr3];
console.log("arr3Copy =", arr3Copy);

// Concatenate arrays
const firstHalf = ['this', 'is'];
const sencondHalf = ['a', 'string'];
const arr5 = [...firstHalf, ...sencondHalf];
console.log("arr5 =", arr5);

// Add entries
const myObj = { a: 1, b: 3 };
const newObj = { c: 5, ...myObj };
console.log("newObj =", newObj);

// Copy and update an entry
const updatedObj = { ...newObj, c: 10 };
console.log("updatedObj =", updatedObj);
```

```
const [first, second, ...rest] = [1, 2, 3, 4, 5, 6];
console.log("rest =", rest);

const { firstObj, ...restObj } = { first: 1, second: 2, third: 3, fourth: 4 };
console.log("restObj =", restObj);

function printArray(...theArgs) {
  console.log("theArgs =", theArgs);
}

printArray(1, 2, 3);
```

ENHANCED OBJECT LITERALS

- **Enhanced object literals**: shorthand syntax for using variables/functions to create object properties where the key matches the variable/function name
  - You can also use computed property names as a key when defining a new object.

```
// ES5
const firstName = 'Name';
const user = {
  firstName: firstName,
  greeting: function greeting() {
    console.log('hi');
  }
};

// ES6
const user1 = {
  firstName,
  greeting() {
    console.log('hi');
  }
};
```

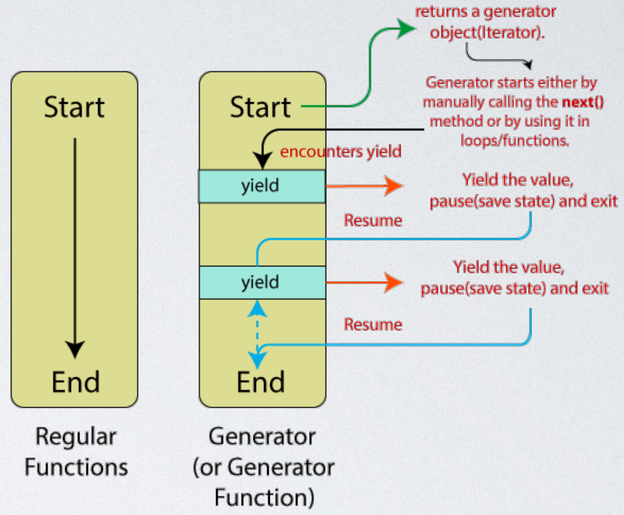
DEFAULT PARAMETERS

- **Default parameters**: function signatures can initialize arguments whose value will be used if nothing or undefined is passed in.

```
const addTen = function (a, b, c = 10) {
  return a + b + c;
};
console.log(addTen(1, 2));
console.log(addTen(1, 2, 20));
console.log(addTen(1, 2, undefined));
```

GENERATORS (ES6)

- **Generator**: an object that controls execution of a generator function.
  - 2 states: **Suspended**, **Closed**
- **Generator function (*)**: a function that can be paused and resumed.
  - When invoked, it returns a new generator instead of executing the code
- **yield**: a unary operator that pauses execution of a generator function and "yields" a value to the generator.
- **.next(arg?)**: a generator method that resumes execution until a value is yielded and returns an object with properties **.value** and **.done**.
  - When execution resumes, arg will replace the yield expression




```
function* genFn() {
  let i = 1;
  console.log(`First yield: ${i}`);
  yield i; // pause execution, yield 1
  i++;
  console.log(`Second yield: ${i}`);
  yield i; // pause execution, yield 2
  i++;
  console.log(`Last yield: ${i}`);
  yield i; // pause execution, yield 3, not done
}

const generator = genFn(); // doesn't execute code in genFn()
console.log("genFn() =", generator); // suspended
console.log("generator.next() =", generator.next()); // execute until first yield
```

OOP

- **Object-oriented programming (OOP)**: a programming paradigm based on the concept of classes, objects, and four main features
  - **Class**: a reusable template for creating objects with specified data properties and methods
  - **Object**: an instantiation of a class

- OOP Features
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

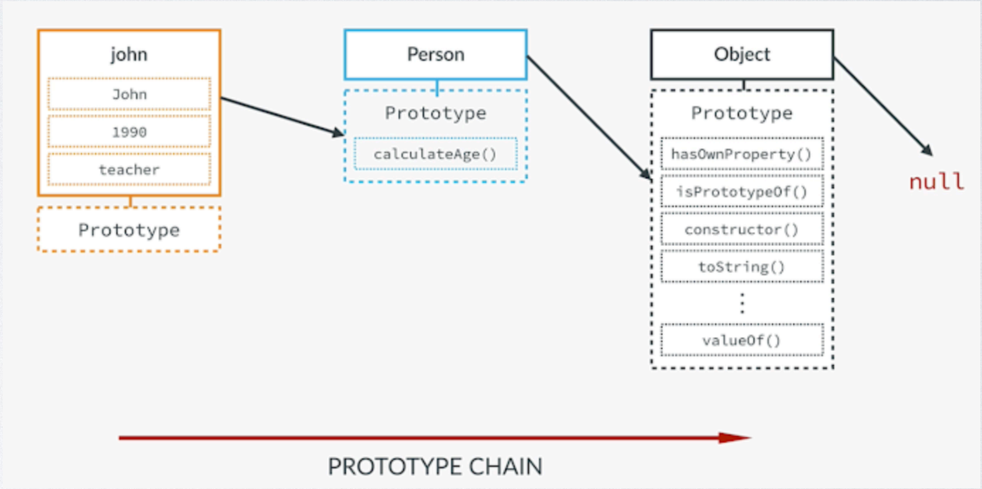
PROTOTYPE-BASED OOP

- **Prototype**: a group of properties and methods that every object "inherits" from.
  - All prototypes have a property .prototype
  - All classes have a property .prototype
  - All objects have a property .__proto__ that is linked to the class.prototype

- ex: Arrays "inherit" from the Array prototype, which defines the array methods

PROTOTYPE CHAIN

- **Prototype chain**: the cycle of looking for an object's property on the current object, then its prototype, and its prototype's prototype, and so on until finding the property or reaching null




```
const arr = [1, 2, 2]; // array literal syntax, creates new Array, arr is instantiated object
console.log("arr.__proto__ === Array.prototype =", arr.__proto__ === Array.prototype);
console.log("arr.__proto__.__proto__ === Object.prototype =",
  arr.__proto__.__proto__ === Object.prototype);
console.log("arr.__proto__.__proto__.__proto__ =", arr.__proto__.__proto__.__proto__);
```

DEFINING & CREATING A CLASS

- **Class keyword (ES6)**: defines templates for creating objects that encapsulate data properties and methods
  - other keywords: extends, static
  - methods: constructor, super (syntactic sugar, built with prototypes)

- **Constructor functions (ES5)**: functions that initialize an object's data properties

- **new**: keyword for creating a new object
  - Used on constructor function or class

- **Object.create(obj)**: creates a new object using an existing object as the prototype

CLASS (ES6)

- Components of a class definition
  - constructor(): a unique method that creates and initializes an object
  - methods

```
// Good practice: name these with upper-case first letter!
class Person {
  constructor(firstname, address) {
    this.firstname = firstname;
    this.address = address;
  }
  say() {
    console.log("class Person say() this =", this);
    console.log(`My name is ${this.firstname}, I live in ${this.address}`);
  }
}
const p1 = new Person('Ethan', 'NJ');
p1.say();
```

- Classes are executed in strict mode

CONSTRUCTOR FUNCTIONS (ES5)

- **new ClassName()**
  - Creates an object where 'this' refers to that object.
  - The new object's property .__proto__ refers to the class.prototype.
  - Return the new object.

- Defining methods: not in the constructor function
  - Declare a separate function and assign it as a property to the Class.prototype

- **hasOwnProperty()**: returns true/false based on whether the object owns this property instead of inheriting it.

```
// Good practice: name these with upper-case first letter!
const Person = function (firstname, address) {
  console.log("Constructor function invoked by new Person(), this =", this);
  this.firstname = firstname;
  this.address = address;
};

Person.prototype.say = function () {
  console.log("Person.prototype.say() this =", this);
  console.log(`My name is ${this.firstname}, I live in ${this.address}`);
};
const p1 = new Person('Ethan', 'NJ');
```

STATIC

- **static**: a keyword for defining static methods and properties on classes, usually for those that do not change across instances
  - Would be accessed through the class (not the instances)

- Logic in static methods is independent of instantiated object state.
  ex: helper methods to public instance methods

```
class PersonES6 {
  constructor(firstname, address) {
    this.firstname = firstname;
    this.address = address;
  }
  static say() {
    console.log('hi');
  }
}
PersonES6.say();

// ----------------------------------------
const PersonES5 = function (firstname, address) {
  this.firstname = firstname;
  this.address = address;
};
PersonES5.say = function () {
  console.log('hi');
};
PersonES5.say();
```

INHERITANCE

- **Inheritance**: an object/class has the properties of another object (prototype-based) or class (class-based) while defining its own properties

- Classes (ES6):
  - **extends** ParentClass
  - **super()**: must be on the first line in the constructor()

- Constructor functions (ES5):
  - Person.call(this, firstname, address)
  - Student.prototype = Object.create(Person.prototype)
  - Student.prototype.constructor = Student

INHERITANCE: ES6 VS ES5

```es6
class PersonES6 {
  constructor(firstname, address) {
    this.firstname = firstname;
    this.address = address;
  }
  say() {
    console.log(`My name is ${this.firstname}, I live in ${this.address}`);
  }
}

class StudentES6 extends PersonES6 {
  constructor(firstname, address, year) {
    // super does the .call()
    super(firstname, address);
    this.year = year;
  }
}
```

```es5
const PersonES5 = function (firstname, address) {
  this.firstname = firstname;
  this.address = address;
};
PersonES5.prototype.say = function () {
  console.log(`My name is ${this.firstname}, I live in ${this.address}`);
};

const StudentES5 = function (firstname, address, year) {
  // execute the parent constructor, where 'this' refers to the student
  // student will be initialized with parent properties
  PersonES5.call(this, firstname, address);
  this.year = year;
};

StudentES5.prototype = Object.create(PersonES5.prototype);
StudentES5.prototype.constructor = StudentES5;
```

THIS (BROWSER)

- **this**: a special keyword that you can use in every function to refer to something.
  - Keyword function 'this' changes based on where it is invoked.
  - Arrow function 'this' refers to the 'this' in the scope where it's defined.
- **Object method**: a function that is a property of an object
- **"use strict"**: makes keyword function 'this' undefined, if keyword function is invoked in any function

In	Keyword function is invoked	Arrow function is defined
Object method	Parent object	Global object
Global scope/ Top-level	Global object	Global object
Keyword function	Global object	Keyword function's this
Arrow function	Global object	Arrow function's this
Event handler	DOM element receiving the event	DOM element receiving the event

CALL, APPLY, BIND

```
const fn = function (num1, num2) {
  console.log("fn this =", this);
  console.log(`${this.num0} | ${num1} | ${num2}`);
};

const obj = {
  num0: 0
};

fn(1, 2);

fn.call(obj, 1, 2);
fn.apply(obj, [1, 2]);
const bindFn = fn.bind(obj);
console.log("bindFn =", bindFn);
bindFn(1, 2);
```

	call()	apply()	bind()
Arguments	this, arg1, …, argN	this, [arg1, …, argN]	this, arg1, …, argN
Effect	Executes function where 'this' refers to the given 'this'	Executes function where 'this' refers to the given 'this'	Copies function and makes 'this' refer to the given 'this'
Return	Result of executing the function	Result of executing the function	The newly-bound function

NEXT MONDAY MOCK

- Today's assignment: due next Monday 9AM EST
- Mock
  - Starts 9:30AM EST
  - Zoom lecture meeting link, breakout rooms, everyone shares screen and cams on, any questions you message us privately
- Coding structure (no google, no references, 1h20m):
  - Leetcode algorithm-style, basic HTML/CSS responsiveness (media queries), DOM manipulation
  - The questions are accessed through the portal => code on your own device (can use VSCode) => upload a .zipped file containing all the solutions
  - We do not accept online editor/sandbox link submissions.
- 15 min break
- SAQ (verbal, just like end of day meetings, no google/references, ~30 min)
  - 25% HTML/CSS, 75% JS
  - Focus more on assignment questions than memorizing all the material from lecture slides
  - May be helpful to create some outline based on assignment SAQ and review it throughout the training

ANY QUESTIONS?