AngularJS

Introduction to AngularJS

Lesson Objectives

- **JavaScript fundamentals**
- **MV* Frameworks**
- **AngularJS Fundamentals**

JavaScript fundamentals

- **Browser gets the HTML text of the page, parses it into DOM structure, lays out the content of the page, and styles the content before it gets displayed.**
- **HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications.**
- **JavaScript has become one of the most popular client side scripting language on the web which is used to create dynamic views in web-applications.**
- **JavaScript plays a major role in the usage of ajax, user experience and responsive web design.**
- **DOM manipulation libraries like jQuery simplifies client side scripting, but it is not solving the problem of handling separation of concerns.**
- **Fortunately there are few libraries and frameworks are available to accomplish this task.**

Objects in JavaScript

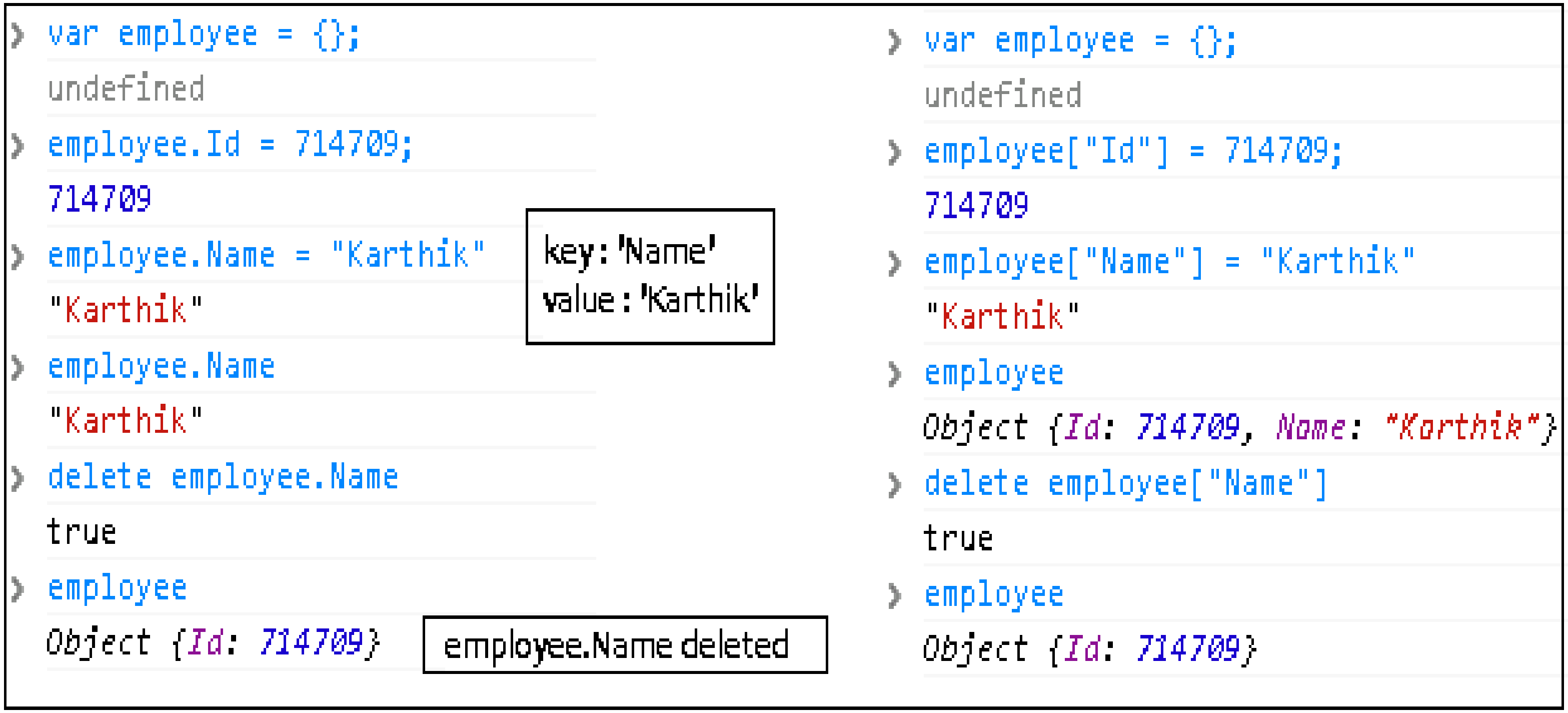
- **JavaScript is an object oriented language. In JavaScript we can define our own objects and assign methods, properties to it.**
- **In JavaScript objects are also associative arrays (or) hashes (key value pairs).**
  - Assign keys with obj[key] = value or obj.name = value
  - Remove keys with delete obj.name
  - Iterate over keys with for(key in obj), iteration order for string keys is always in definition order, for numeric keys it may change.
- **Properties, which are functions, can be called as obj.method(). They can refer to the object as *this*. Properties can be assigned and removed any time.**
- **A function can create new objects when run in constructor mode as new Func(params). Names of such functions are usually capitalized**

Creating objects

- **An empty object can be creating using**
  - obj = new Object();   (or)   obj = { };
  - It stores values by key, with that we can assign or delete it using "dot notation" or "Square Brackets" (associative arrays).

using dot notation

using square brackets



```
› var employee = {};
  undefined
› employee.Id = 714709;
  714709
› employee.Name = "Karthik"
  "Karthik"
› employee.Name
  "Karthik"
› delete employee.Name
  true
› employee
  Object {Id: 714709}
```

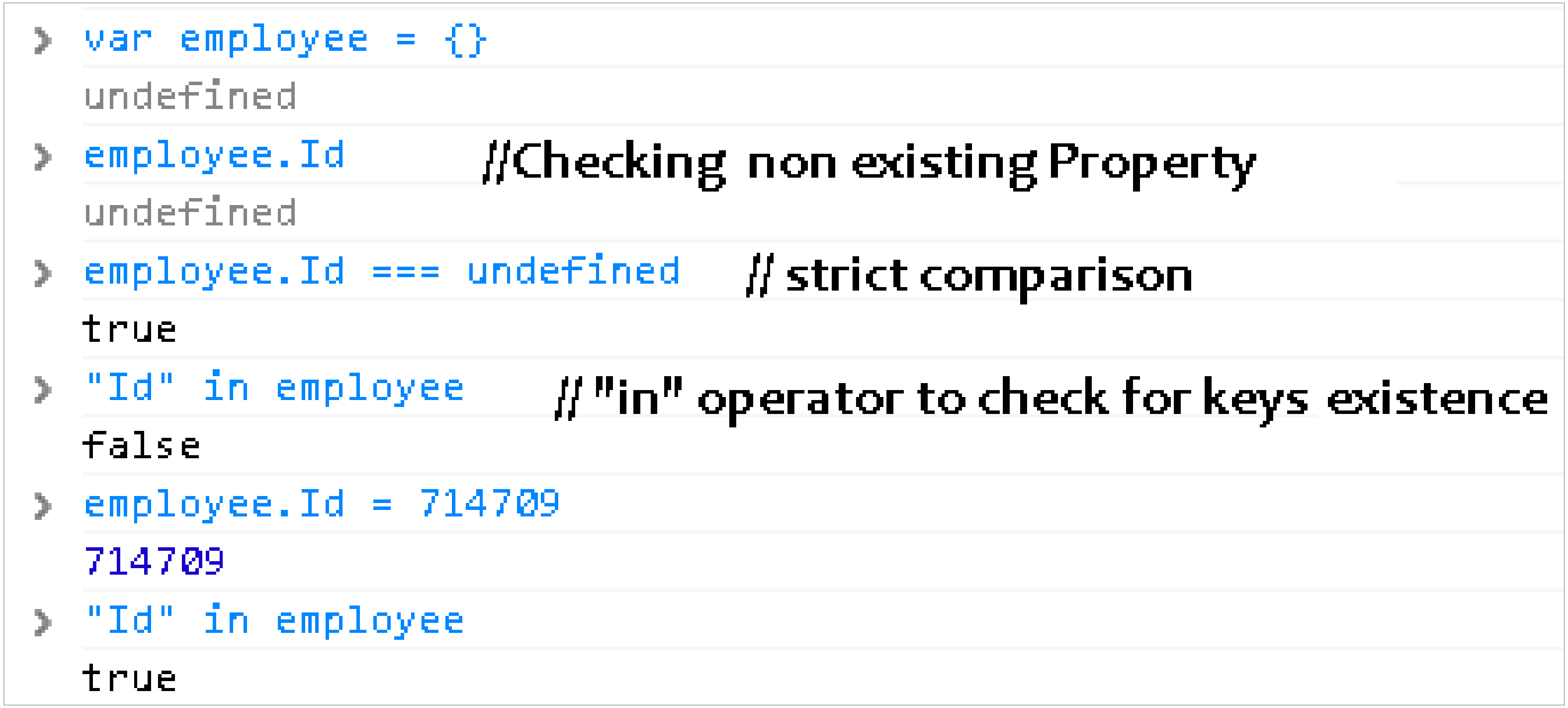
key: 'Name'
value: 'Karthik'

employee.Name deleted

```
› var employee = {};
  undefined
› employee["Id"] = 714709;
  714709
› employee["Name"] = "Karthik"
  "Karthik"
› employee
  Object {Id: 714709, Name: "Karthik"}
› delete employee["Name"]
  true
› employee
  Object {Id: 714709}
```

Checking for non existing property in object

- If the property does not exist in the object , then *undefined* is returned
- To check whether key existence we can use *in* operator

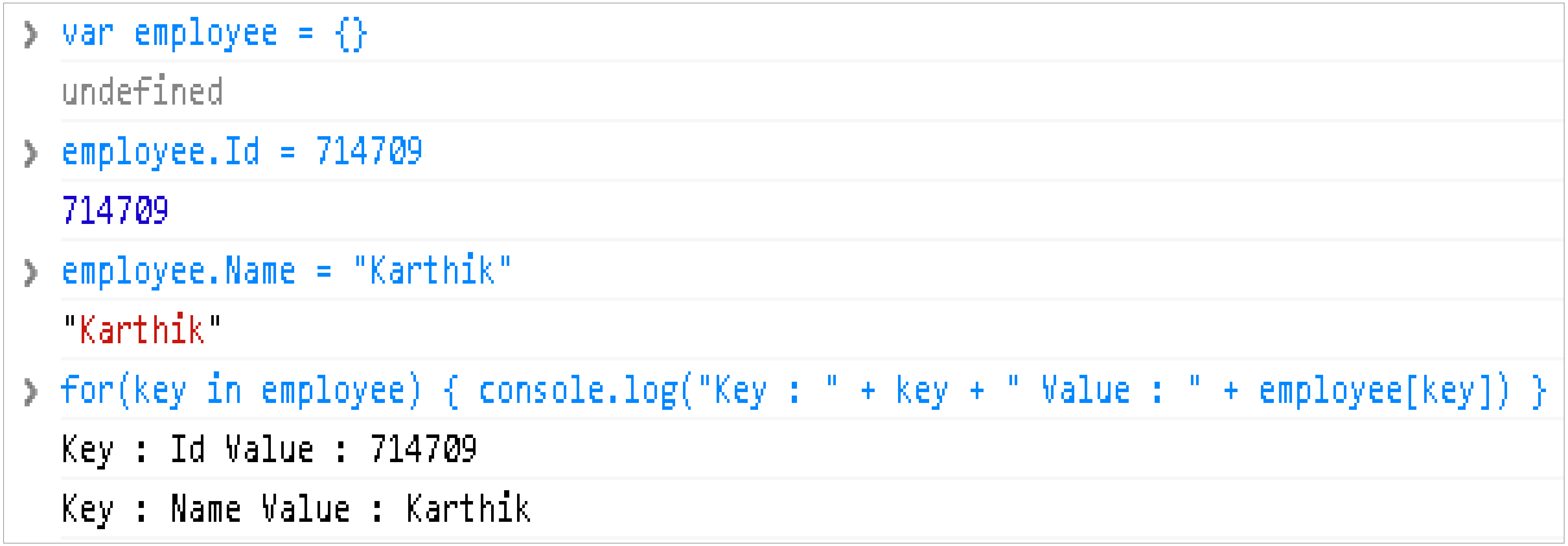


```
> var employee = {}
  undefined
> employee.Id           //Checking non existing Property
  undefined
> employee.Id === undefined     // strict comparison
  true
> "Id" in employee        // "in" operator to check for keys existence
  false
> employee.Id = 714709
  714709
> "Id" in employee
  true
```

IGATE
Speed.Agility.Imagination

Iterating over object keys

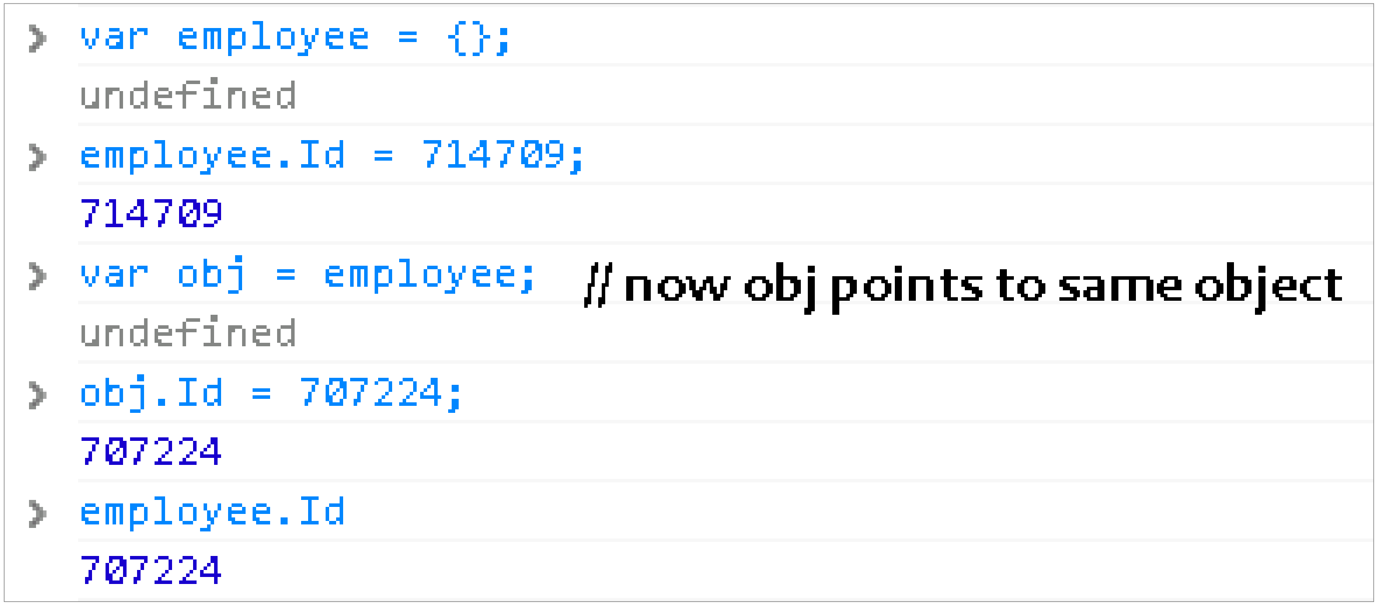
- **We can iterate over keys using *for .. In***



```
> var employee = {}
  undefined
> employee.Id = 714709
  714709
> employee.Name = "Karthik"
  "Karthik"
> for(key in employee) { console.log("Key : " + key + " Value : " + employee[key]) }
  Key : Id Value : 714709
  Key : Name Value : Karthik
```

Object reference

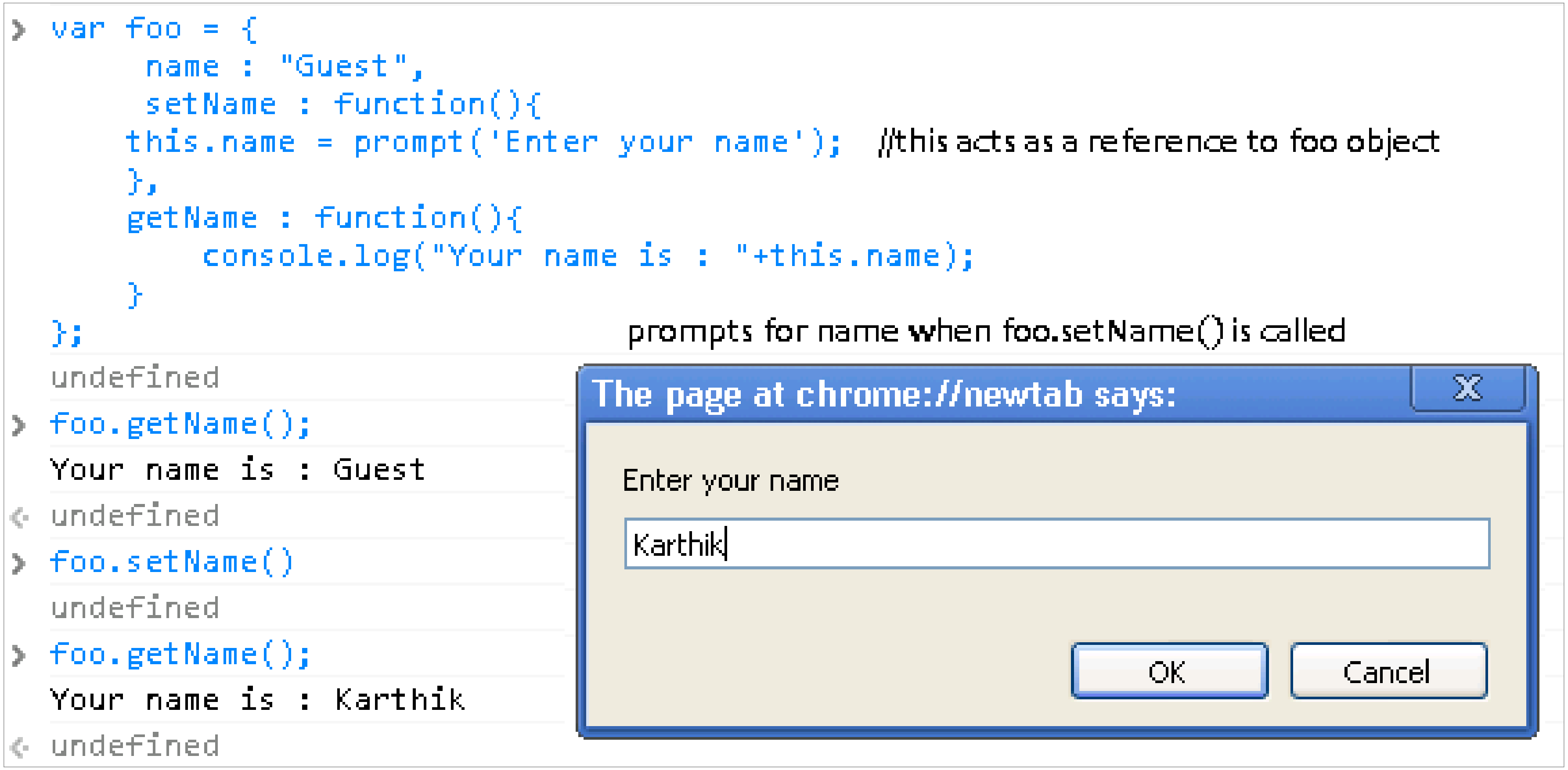
- **A variable which is assigned to object actually keeps reference to it.**
- **It acts like a pointer which points to the real data. Using reference variable we can change the properties of object.**
- **Variable is actually a reference, not a value when we pass an object to a function.**



```
> var employee = {};
  undefined
> employee.Id = 714709;
  714709
> var obj = employee;     // now obj points to same object
  undefined
> obj.Id = 707224;
  707224
> employee.Id
  707224
```

this keyword

- **When a function is called from the object, *this* becomes a reference to this object.**



```
> var foo = {
       name : "Guest",
       setName : function(){
      this.name = prompt('Enter your name');    //this acts as a reference to foo object
      },
      getName : function(){
          console.log("Your name is : "+this.name);
      }
  };
  undefined
> foo.getName();
  Your name is : Guest
< undefined
> foo.setName()
  undefined
> foo.getName();
  Your name is : Karthik
< undefined
```

prompts for name when foo.setName() is called

The page at chrome://newtab says:

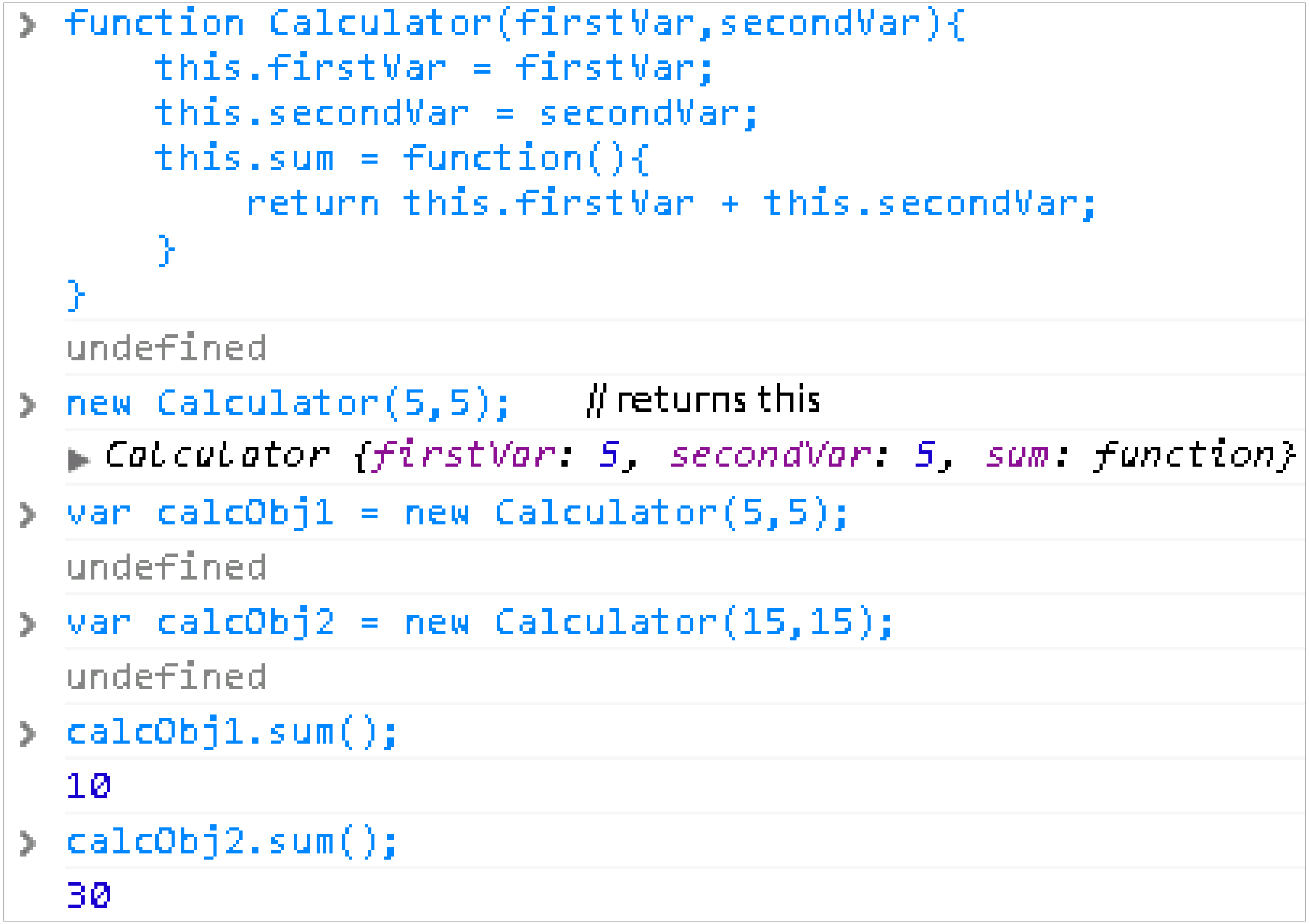
Enter your name

Karthik

Constructor Function

- **We can create an object using obj = {……}**
- **Another way of creating an object in JavaScript is to construct it by calling a function with new directive (Constructor function). Constructor functions should be in Pascal case.**
- **It takes this, which is initially an empty object, and assigns properties to it. The result is returned (unless the function has explicit return).**

Constructor Function

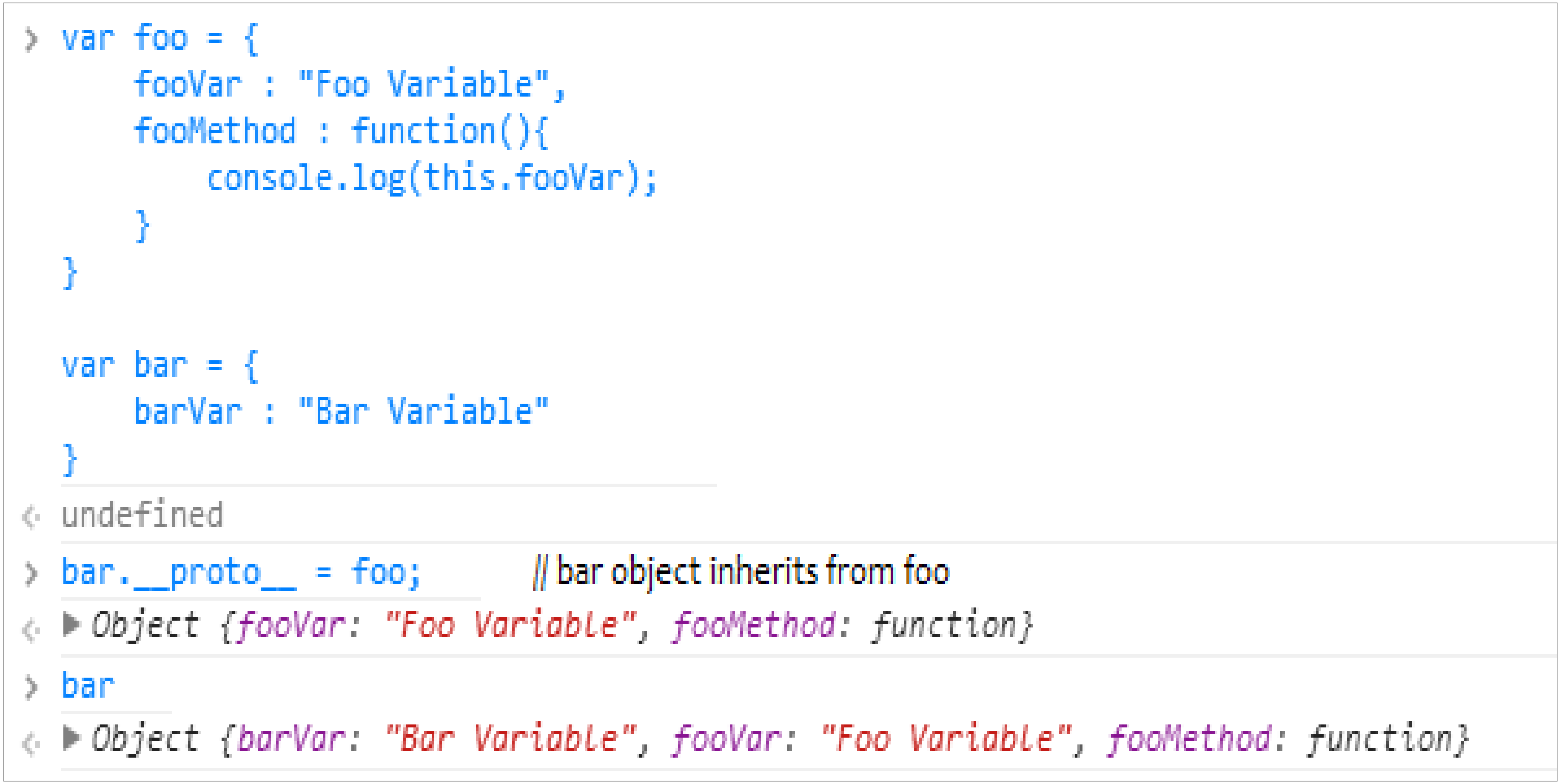


```
> function Calculator(firstVar,secondVar){
      this.firstVar = firstVar;
      this.secondVar = secondVar;
      this.sum = function(){
          return this.firstVar + this.secondVar;
      }
  }
  undefined
> new Calculator(5,5);     // returns this
  ▶ Calculator {firstVar: 5, secondVar: 5, sum: function}
> var calcObj1 = new Calculator(5,5);
  undefined
> var calcObj2 = new Calculator(15,15);
  undefined
> calcObj1.sum();
  10
> calcObj2.sum();
  30
```

Prototypal inheritance

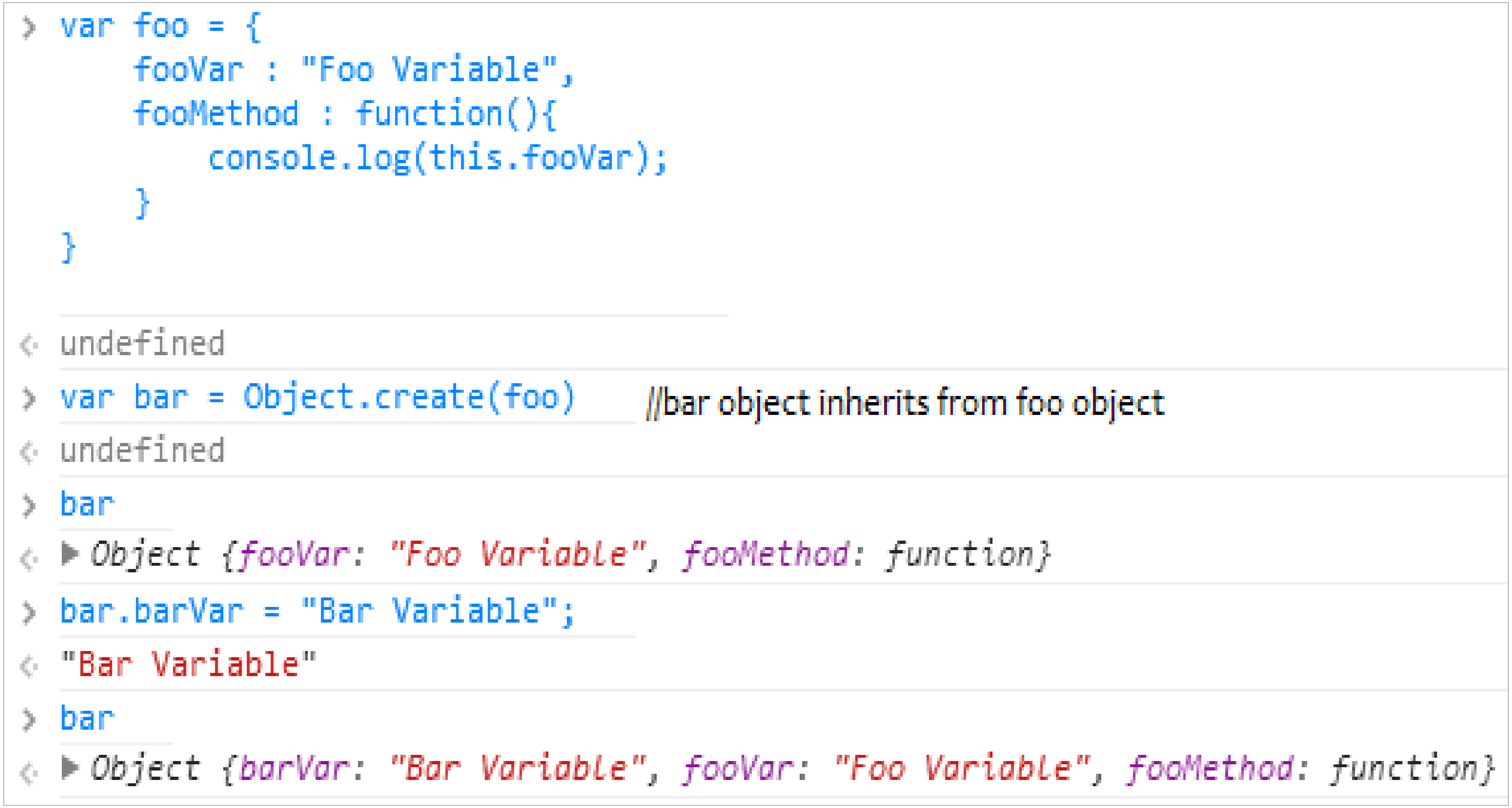
- **In JavaScript, the inheritance is prototype-based. Instead of class inherits from other class, an object inherits from another object.**
- **object inherits from another object using the following syntax.**
- ***childObject.__proto__ = baseObject***
  - Above mentioned syntax provided by Chrome / FireFox. In other browsers the property still exists internally, but it is hidden
- ***childObject = Object.create(baseObject)***
- ***ConstructorFunction.prototype = baseObject***
  - Above mentioned syntax works with all modern browsers.

Prototypal inheritance using __proto__



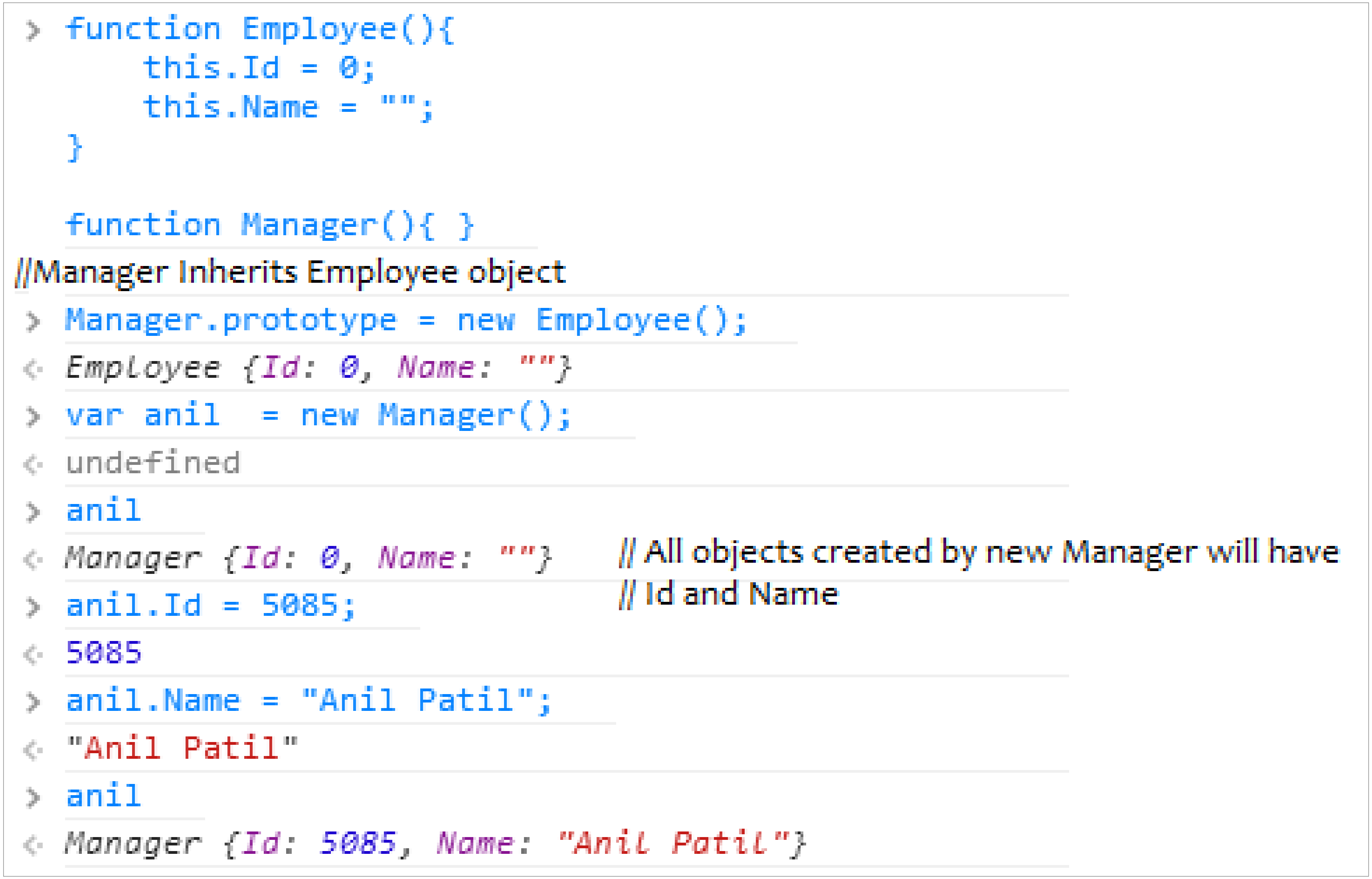
```
> var foo = {
      fooVar : "Foo Variable",
      fooMethod : function(){
          console.log(this.fooVar);
      }
  }

  var bar = {
      barVar : "Bar Variable"
  }
< undefined
> bar.__proto__ = foo;        // bar object inherits from foo
< ▶ Object {fooVar: "Foo Variable", fooMethod: function}
> bar
< ▶ Object {barVar: "Bar Variable", fooVar: "Foo Variable", fooMethod: function}
```

Prototypal inheritance using Object.create()



```
> var foo = {
      fooVar : "Foo Variable",
      fooMethod : function(){
          console.log(this.fooVar);
      }
  }

< undefined
> var bar = Object.create(foo)       //bar object inherits from foo object
< undefined
> bar
< ▶ Object {fooVar: "Foo Variable", fooMethod: function}
> bar.barVar = "Bar Variable";
< "Bar Variable"
> bar
< ▶ Object {barVar: "Bar Variable", fooVar: "Foo Variable", fooMethod: function}
```

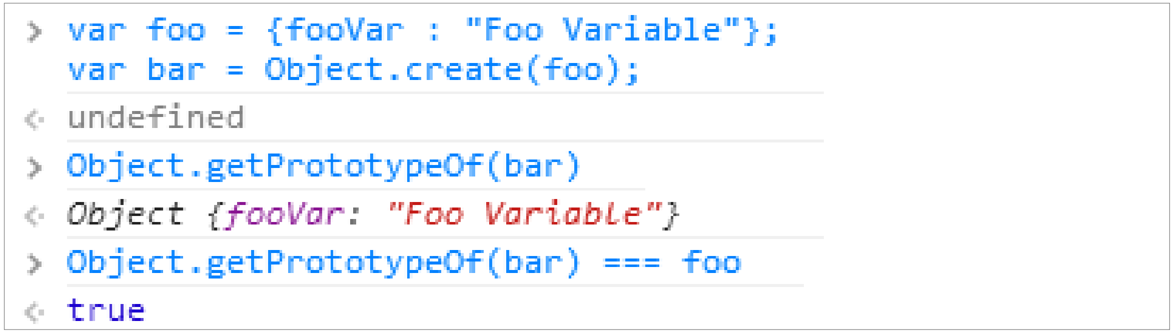
Prototypal inheritance using prototype



```
> function Employee(){
        this.Id = 0;
        this.Name = "";
  }

  function Manager(){ }
//Manager Inherits Employee object
> Manager.prototype = new Employee();
< Employee {Id: 0, Name: ""}
> var anil  = new Manager();
< undefined
> anil
< Manager {Id: 0, Name: ""}        // All objects created by new Manager will have
> anil.Id = 5085;                    // Id and Name
< 5085
> anil.Name = "Anil Patil";
< "Anil Patil"
> anil
< Manager {Id: 5085, Name: "Anil Patil"}
```

Prototypal inheritance

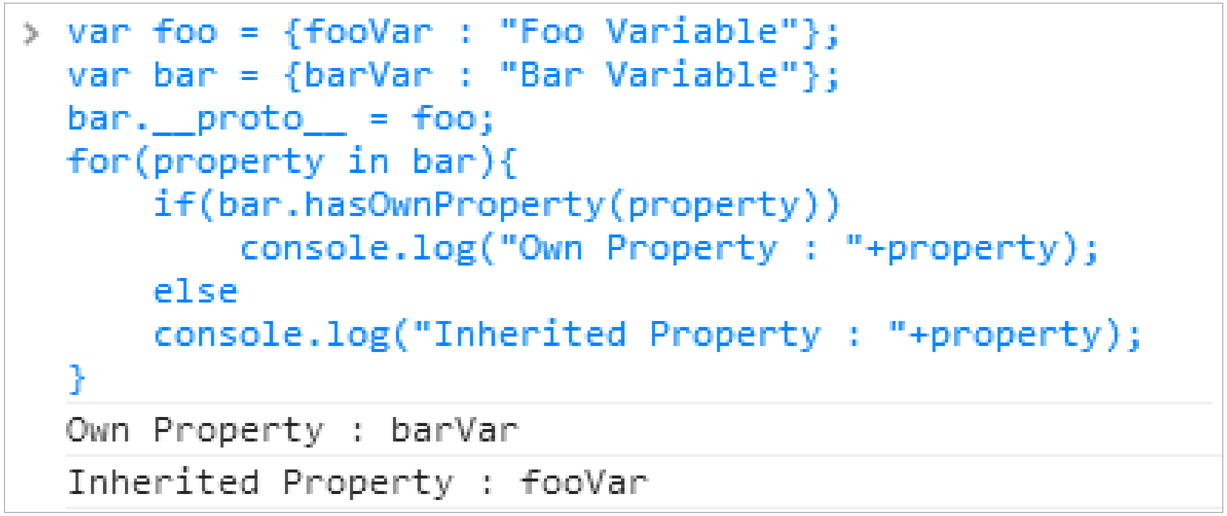
- ***Object.getPrototypeOf(obj)*** **returns the value of obj.__proto__.**



```
> var foo = {fooVar : "Foo Variable"};
  var bar = Object.create(foo);
< undefined
> Object.getPrototypeOf(bar)
< Object {fooVar: "Foo Variable"}
> Object.getPrototypeOf(bar) === foo
< true
```

- **for..in loop lists properties in the object and its prototype chain.**

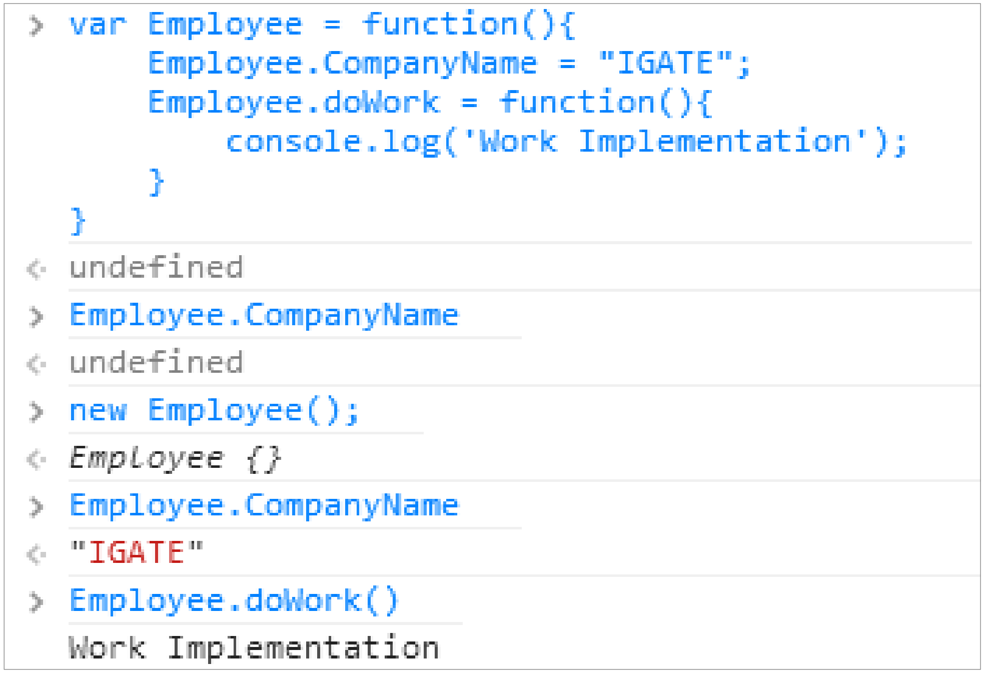
  obj.hasOwnProperty(prop) returns true if property belongs to that object.



```
> var foo = {fooVar : "Foo Variable"};
  var bar = {barVar : "Bar Variable"};
  bar.__proto__ = foo;
  for(property in bar){
      if(bar.hasOwnProperty(property))
          console.log("Own Property : "+property);
      else
      console.log("Inherited Property : "+property);
  }
  Own Property : barVar
  Inherited Property : fooVar
```

Static variables and methods

- In JavaScript we can directly put data into function object which acts like Static member.
- Static Members need to be accessed directly by Object name, cannot be accessed by reference variable. Static members gets created when the first object gets created.



```
> var Employee = function(){
      Employee.CompanyName = "IGATE";
      Employee.doWork = function(){
          console.log('Work Implementation');
      }
  }
< undefined
> Employee.CompanyName
< undefined
> new Employee();
< Employee {}
> Employee.CompanyName
< "IGATE"
> Employee.doWork()
  Work Implementation
```

JavaScript Functions

- **JavaScript treats functions as objects(first-class functions).**
- **In JavaScript functions can be instantiated, returned by other functions, stored as elements of arrays and assigned to variables.**
- **A function with no name is called an anonymous function.**
- **Closure is a function to which the variables of the surrounding context are bound by reference.**
- **JavaScript function acts as a constructor when we use it together with the new operator**

Working with JavaScript Functions

- **Declaring the function anonymously**

```
function(){
        console.log(‘IGATE');
}
```

- **Invoking the anonymous function. Function executes immediately after declaration.**

```
(function(){
        console.log(‘IGATE');
})();
```

Working with JavaScript Functions

- **Declaring a named function. function doSomething will be available inside the scope in which it’s declared.**

```
function doSomething(){
        console.log('IGATE');
}

/* Inner Scope */
(function(){
        doSomething();
})();
```

- **Assigning function to a variable.**

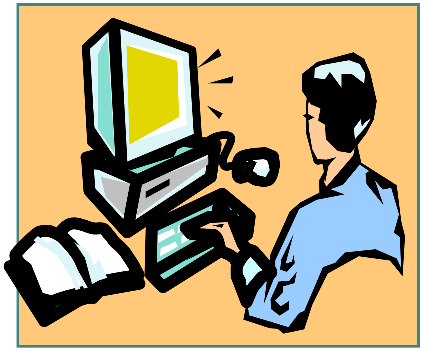
```
var doSomething = function(){
        console.log(‘IGATE');
}
```

Working with JavaScript Functions

```
/*Anonymous Closures*/
(function(){
        var data = "Closing the variables inside the function from the rest of
the world"
        console.log(‘Closure Invoked');
})();

var employee = function(){
        this.employeeId = 0;
        this.name = "";
};
/* JavaScript function acts as a constructor */
var emp = new  employee();
```

Demo

- **Working-with-Javascript-Functions**
- **Closure-Demo**



MV* Frameworks

- **MV* Frameworks are designed to make our code easier to maintain and to improve the user experience**
- **MV* framework is nothing but the popular patterns like**
  - Model-View-Controller(MVC)
  - Model-View-ViewModel(MVVM)
  - Model-View-Presenter(MVP)
  - MVW(hatever works for you)
- **Idea of all the patterns is to separate Model, View and the Controller (the logic that hooks up model and view)**
- **AngularJS, Backbone.JS, Knockout, EmberJS, Meteor, ExtJS are some of the famous MV* framework libraries.**

Model, View and Controllers

Model

- Contains the data which we are using in our application

View

- Displays the data to the user and read the user input.

Controller

- Format the data for views and handle application state.

Introduction to AngularJS

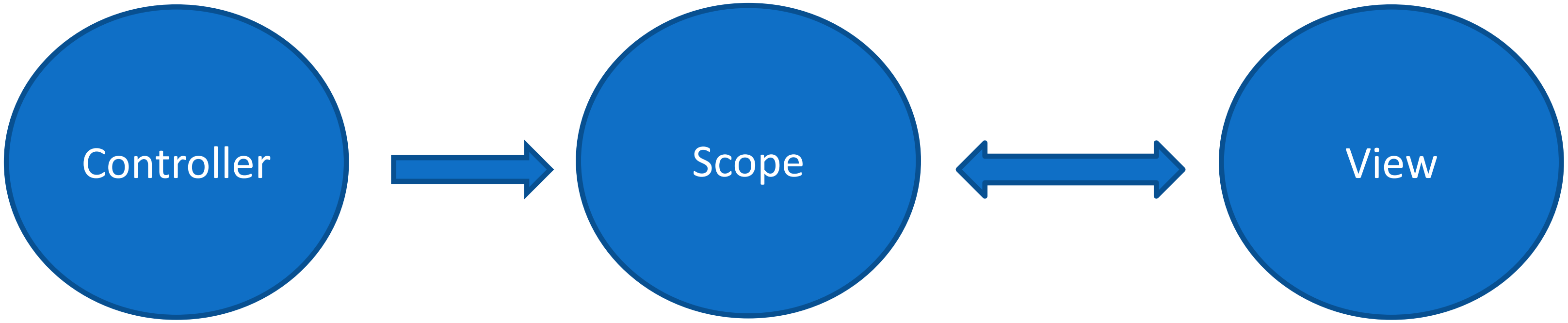
- **AngularJS is an open source JavaScript library that is sponsored and maintained by Google.**
- **Developed in 2009 by Misko Hevery. Publicly released as version 0.9.0 in Oct 2010.**
- **AngularJS makes it easy to build interactive, modern web applications by increasing the level of abstraction between the developer and common web app development tasks by following Model–View–Controller (MVC) pattern.**
- **AngularJS lets you to extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.**
- **AngularJS helps us to create single page applications easily.**
  - No page refresh on page change and different data on each page

AngularJS Features

- **Extending HTML to add dynamic nature so that we can build modern web applications with separation of application logic, data models, and views templates**
- **Two way binding**
  - It synchronize the data between model and view, view component gets updated when the model get change and vice versa, no need for events to accomplish this
- **Templates can be created using HTML itself**
- **Testability is the primary consideration in AngularJS. It supports both isolated unit tests and Integrated end to end test**
- **It also supports Routing, Filtering, Ajax calls, data binding, caching, history, and DOM manipulation.**

AngularJS Controller and Scope

- **Controllers primary responsibility is to create scope object ($scope), It also constructs the model on $scope and provides commands for the view to act upon $scope.**
- **Scope communicate with view in two way communication**
- **Scope exposes model to view, but scope is not a model. Model is nothing but the data present in the scope.**
- **View can be binded to the functions on the scope.**
- **We can modify the model using the methods available on the scope.**



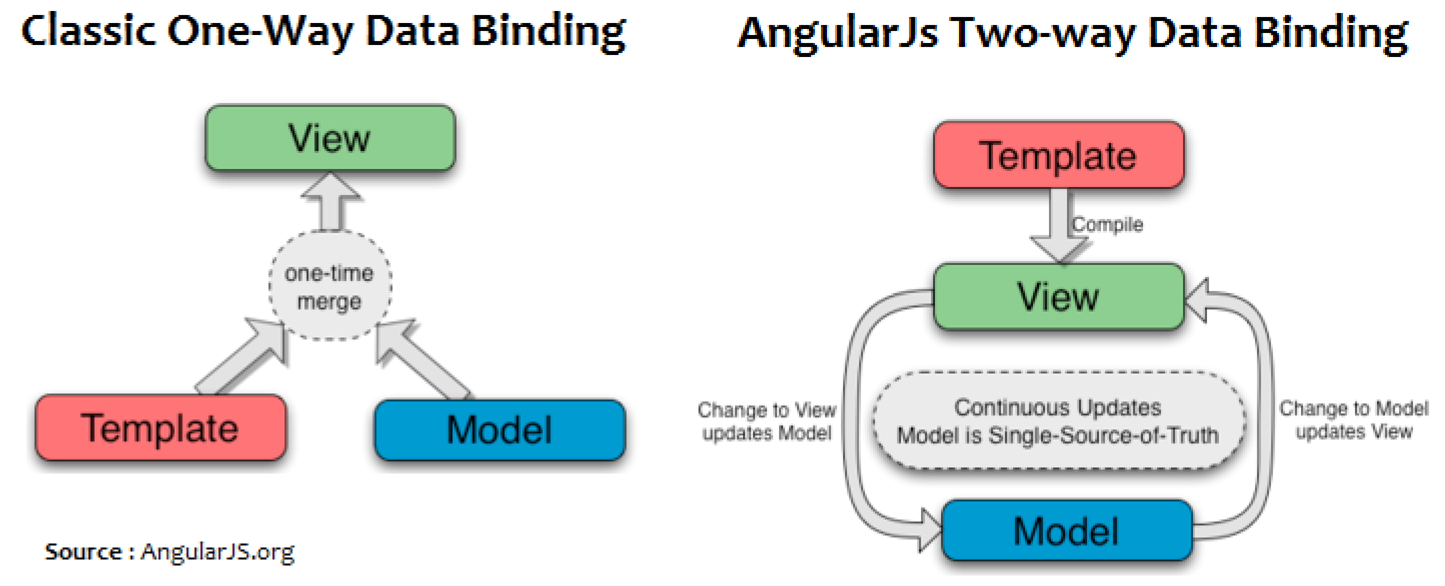
$scope is the glue between Controller and Model

AngularJS Model

- **The model is simply a plain old JavaScript object, does not use getter/setter methods or have any special framework-specific needs.**
- **Changes are immediately reflected in the view via the two-way binding feature.**
- **All model objects stem from scope object.**
- **Typically model objects are initialized in controller code with syntax like:**
  - *$scope.companyName = "IGATE";*
- **In the HTML template, that model variable would be referenced in curly braces such as: {{companyName}} without the $scope prefix.**

AngularJS View and Templates

- **The View in AngularJS is the compiled DOM.**
- **View is the product of $compile merging the HTML template with $scope.**
- **In Angular, templates are written with HTML that contains Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser**


Classic One-Way Data Binding

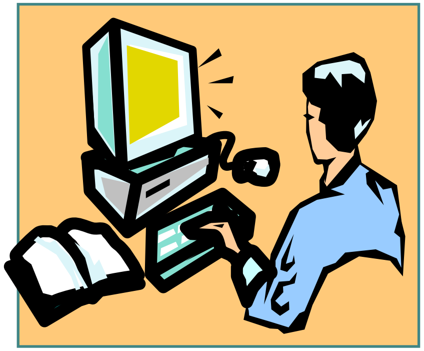
AngularJs Two-way Data Binding

View
one-time merge
Template
Model
Template
Compile
View
Change to View updates Model
Continuous Updates Model is Single-Source-of-Truth
Change to Model updates View
Model

Source : AngularJS.org

Demo

- **AngularJs-MVC**



AngularJS Modules

- **A module is the overall container used to group AngularJS code. It consists of compiled services, directives, views controllers, etc.**
- **Module is like a main method that instantiates and wires together the different parts of the application.**
- **Modules declaratively specify how an application should be bootstrapped.**
- **The Angular module API allows us to declare a module using the angular.module() API method.**
- **When declaring a module, we need to pass two parameters to the method. The first is the name of the module we are creating. The second is the list of dependencies, otherwise known as injectables.**
  - angular.module('myApp', []); // setter method for defining Angular Module.
  - angular.module('myApp'); // getter method for referencing Angular Module.

AngularJs Expressions

- Expressions {{expression}} are JavaScript like code snippets.
- In Angular, expressions are evaluated against a scope object.
- AngularJS let us to execute expressions directly within our HTML pages.
- Expressions are generally placed inside a binding and typically it has variable names set in the scope object.
- Expression can also hold computational codes like {{3 * 3}}, but we cannot directly use JavaScript syntax like {{Math.random()}}, conditionals, loops or exceptions inside it.

```
<div>{{3 * 3}}</div> returns 9

<div>{{'Karthik'+' '+'Muthukrishnan'}}</div> returns  Karthik Muthukrishnan

<div>{{['Ganesh','Abishek','Karthik','Anil'][2]}}</div> returns Karthik
```

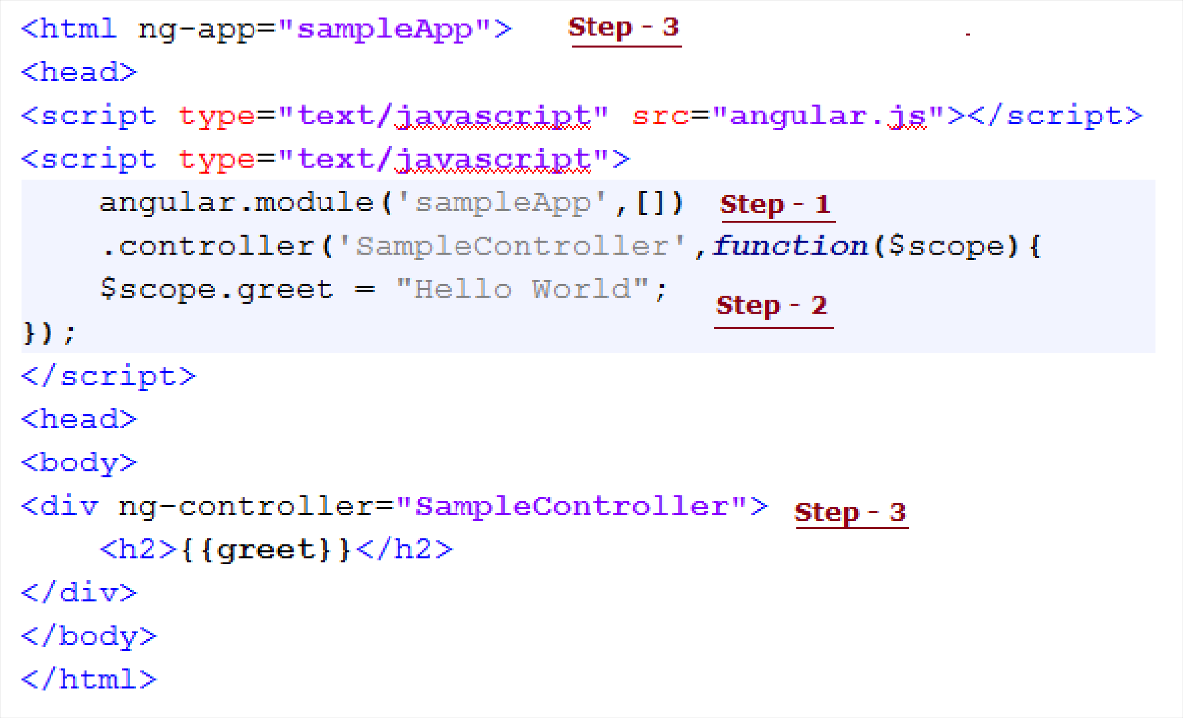
$rootScope

- When Angular starts to run and generate the view, it will create a binding from the root ng-app element to the $rootScope.
- $rootScope is the eventual parent of all $scope objects and it is set when the module initializes via run method.
- The $rootScope object is the closest object we have to the global context in an Angular app. It's a bad idea to attach too much logic to this global context.

```
<div ng-app="myApp">     <h1>{{companyName}}</h1>   </div>
<script>
        var app= angular.module("myApp",[]);
        app.run(function($rootScope){
                $rootScope.companyName = "IGATE";
                $rootScope.printCompanyName = function() {
                        console.log($rootScope.companyName);
                }
        });
</script>
```

Steps for Coding Hello World in AngularJs

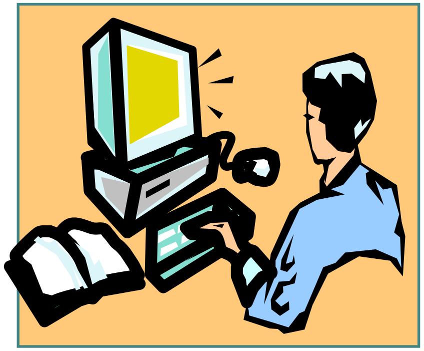
- **Step 1: Declare the module**
- **Step 2: Declare the controller and set the properties (or) function to the scope.**
- **Step 3: Bootstrap angularjs using *ng-app* and define the controller, so that the properties which we have set in the controller can be consumed in the view(HTML).**



```
<html ng-app="sampleApp">    Step - 3
<head>
<script type="text/javascript" src="angular.js"></script>
<script type="text/javascript">
    angular.module('sampleApp',[])  Step - 1
    .controller('SampleController',function($scope){
    $scope.greet = "Hello World";
});                                  Step - 2
</script>
<head>
<body>
<div ng-controller="SampleController">  Step - 3
    <h2>{{greet}}</h2>
</div>
</body>
</html>
```

Demo

- **AngularJs-Modules**



Dependency Injection

- **Dependency injection is a design pattern that allows for the removal of hard-coded dependencies, thus making it possible to remove or change them at run time.**

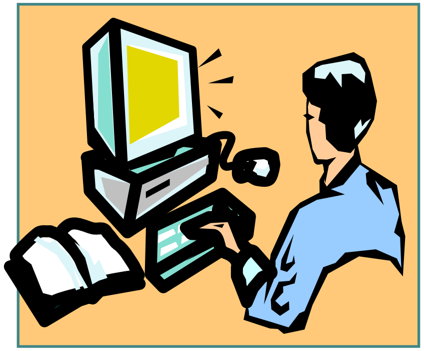
```
function Foo(object) {
   this.object = object;
}
Foo.prototype.showDetails = function(data) {
  this.object.display(data);
}

var greeter = {
        display : function(msg){
          alert(msg);
        }
}

var foo = new Foo(greeter);
foo.showDetails("IGATE");
```

Dependency Injection

- **At runtime, the Foo doesn’t care how it gets the dependency, so long as it gets it. In order to get that dependency instance into Foo, the creator of Foo is responsible for passing in the Foo dependencies when it’s created.**
- **This ability to modify dependencies at run time allows us to create isolated environments that are ideal for testing. We can replace real objects in production environments with mocked ones for testing environments.**
- **In AngularJS at run time, an injector will create instances of the dependencies and pass them along to the dependent consumer.**

Demo

- **Dependency Demo**



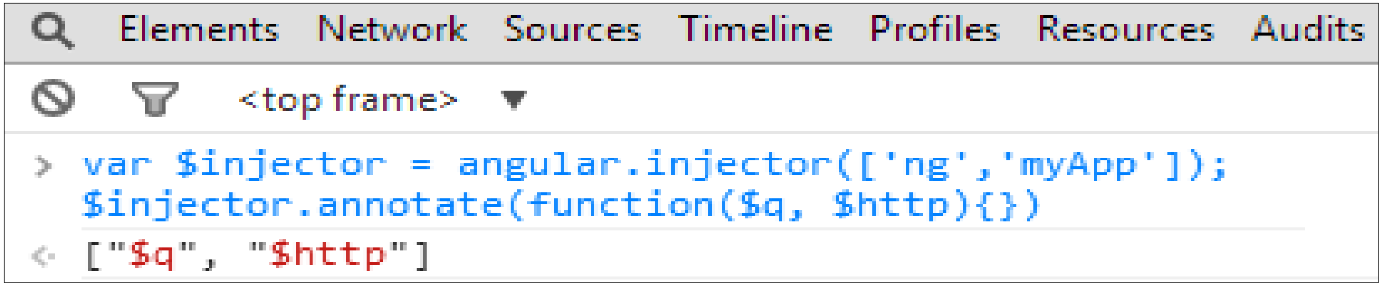
AngularJS Services

- **Services provide a method for us to keep data around for the lifetime of the app and communicate across controllers in a consistent manner.**
- **Services are singleton objects that are instantiated only once per app (by the $injector) and lazyloaded (created only when necessary).**
- **We need to put our business logic in Services.**

We will discuss in detail about Services, later in this course.

injector Service

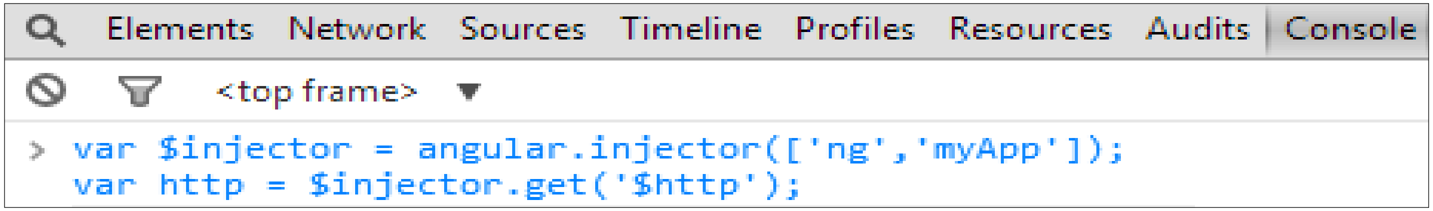
- **Angular uses the injector for managing lookups and instantiation of dependencies. We will very rarely work directly with injector service**
- **injector is responsible for handling all instantiations of our Angular components, including app modules, directives, controllers, etc.**
- **injector is responsible for instantiating the instance of the object and passing in any of its required dependencies. Injector API has following methods.**
- **annotate()**
  - The annotate() function returns an array of service names that are to be injected into the function when instantiated.



```
> var $injector = angular.injector(['ng','myApp']);
  $injector.annotate(function($q, $http){})
< ["$q", "$http"]
```

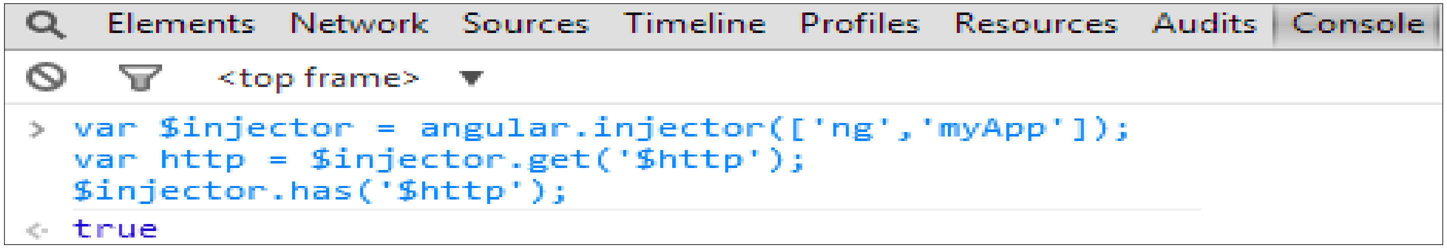
injector Service

- **get()**
  - The get() method returns an instance of the service which takes the name argument. (the name of the instance we want to get).



```
> var $injector = angular.injector(['ng','myApp']);
  var http = $injector.get('$http');
```

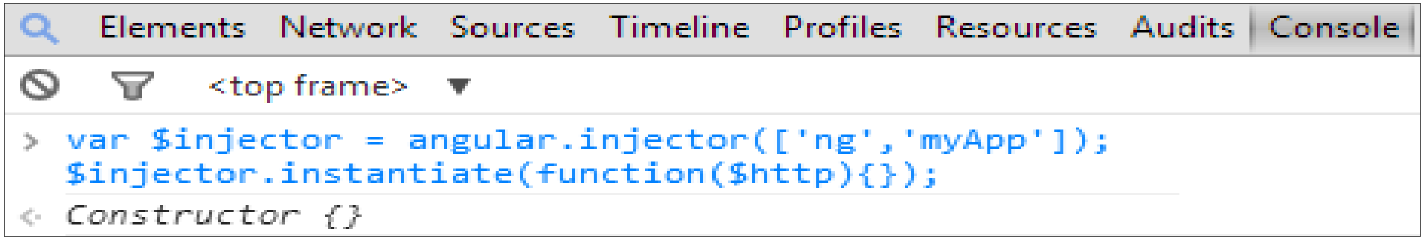
- **has()**
  - The has() method returns true if the injector knows that a service exists in its registry and false if it does not.



```
> var $injector = angular.injector(['ng','myApp']);
  var http = $injector.get('$http');
  $injector.has('$http');
< true
```

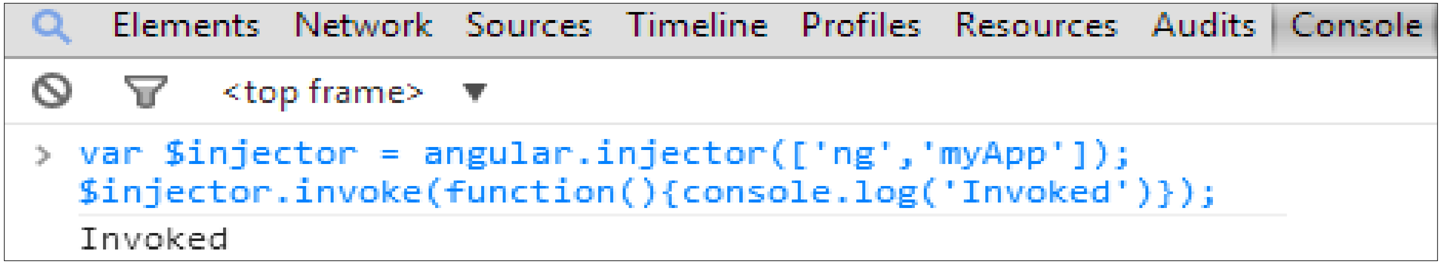
injector Service

- **instantiate()**
  - The instantiate() method creates a new instance of the JavaScript type. It takes a constructor and invokes the new operator with all of the arguments specified.



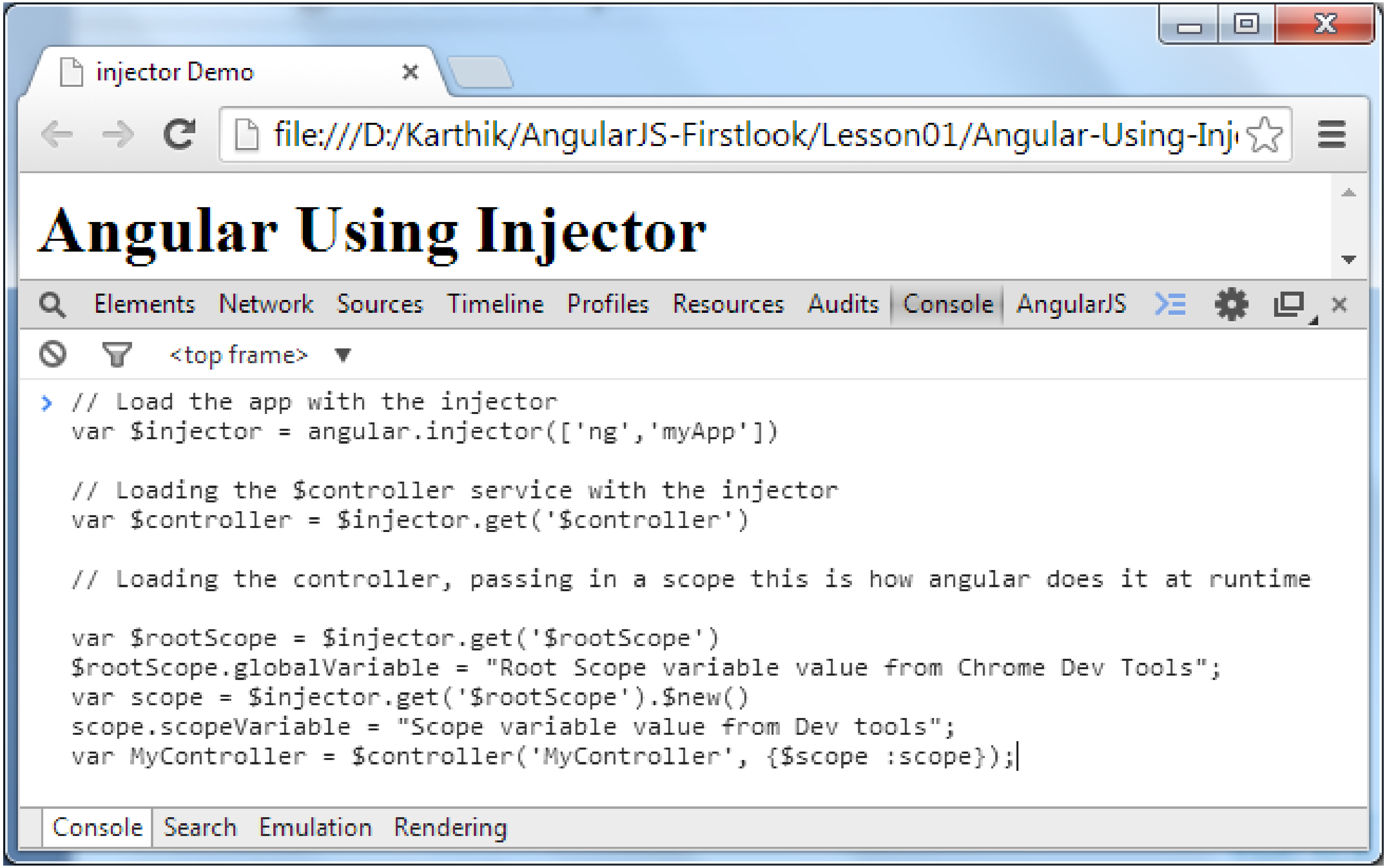
```
> var $injector = angular.injector(['ng','myApp']);
  $injector.instantiate(function($http){});
< Constructor {}
```

- **invoke()**
  - The invoke() method invokes the method and adds the method arguments from the $injector. The invoke() method returns the value that the fn function returns



```
> var $injector = angular.injector(['ng','myApp']);
  $injector.invoke(function(){console.log('Invoked')});
  Invoked
```

How Angular uses injector Service

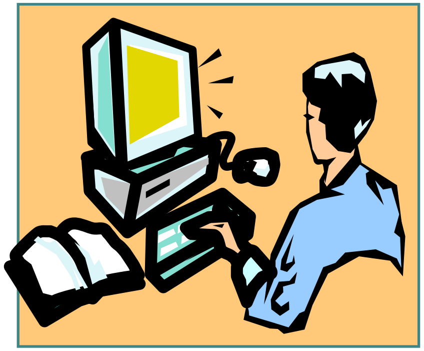

injector Demo
file:///D:/Karthik/AngularJS-Firstlook/Lesson01/Angular-Using-Inj
Elements Network Sources Timeline Profiles Resources Audits Console AngularJS
<top frame>
Console Search Emulation Rendering

Angular Using Injector

```
// Load the app with the injector
var $injector = angular.injector(['ng','myApp'])

// Loading the $controller service with the injector
var $controller = $injector.get('$controller')

// Loading the controller, passing in a scope this is how angular does it at runtime

var $rootScope = $injector.get('$rootScope')
$rootScope.globalVariable = "Root Scope variable value from Chrome Dev Tools";
var scope = $injector.get('$rootScope').$new()
scope.scopeVariable = "Scope variable value from Dev tools";
var MyController = $controller('MyController', {$scope :scope});
```

Demo

- **Angular-Using-Injector**
- **Injector-Demo**

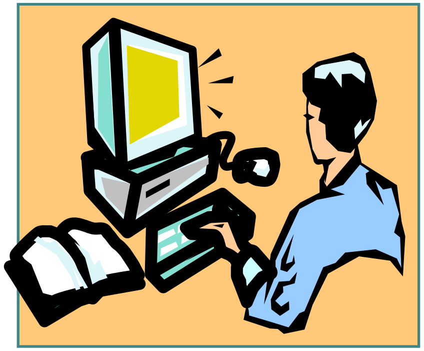


Config and Run Method

- **angular.Module type has config() and run() method**
- **config(configFn)**
  - We can use this method to register the work which needs to be performed on module loading.
  - It will be very useful for configuring the service.
- **run(initializationFn)**
  - We can use this method to register the work which needs to be performed when the injector is done loading all modules.

Demo

- **Config-Demo**

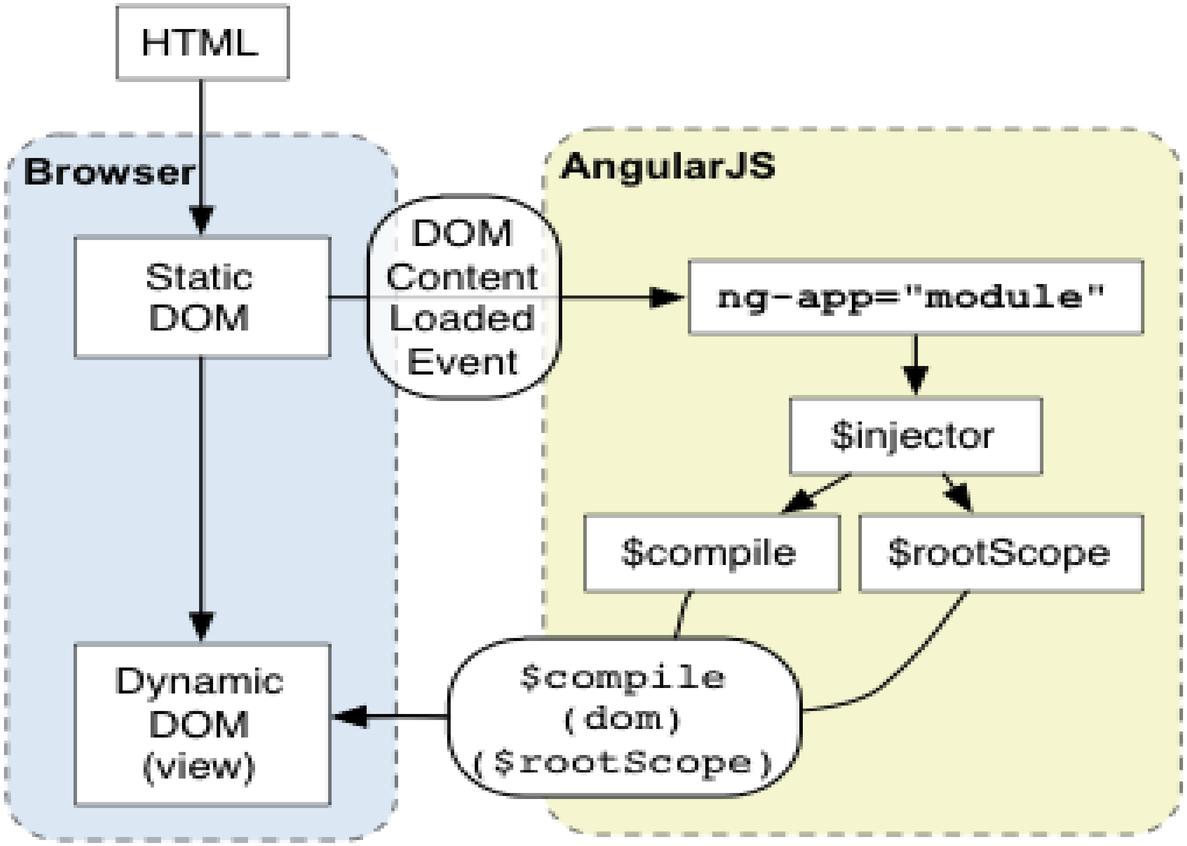


jqLite

- **Angular.js comes with a simple compatible implementation of jQuery called jqLite**
- **Angular doesn’t depend on jQuery. In order to keep Angular small, Angular implements only a subset of the selectors in jqLite, so error will occurs when a jqLite instance is invoked with a selector other than this subset.**
- **We can include a full version of jQuery, which Angular will automatically use. So that all the selectors will be available.**
- **If jQuery is available, *angular.element* is an alias for the jQuery function. If jQuery is not available, *angular.element* delegates to Angular's built-in subset of jQuery, called "jQuery lite" or "jqLite."**
- **All element references in Angular are always wrapped with jQuery or jqLite; they are never raw DOM references.**

How AngularJs Works

- **$compile compiles DOM into a template function that can be used to link scope and the view together.**



Source : Angularjs.org

Summary

- JavaScript objects are also associative arrays
- In JavaScript, the inheritance is prototype-based.
- JavaScript treats functions as objects(first-class functions).
- JavaScript function acts as a constructor when we use it together with the new operator.
- Angular thinks of HTML as if it had been designed to build applications instead of documents.
- Angular supports unit tests and end to end tests.
- Controller is the central component in an angular application.

Summary

- Creating the scope is the primary responsibility of the controller.
- $scope is the glue between Controller and Model
- View can bind to the properties as well as functions on the scope.
- Expressions only supports a subset of JavaScript. We can create an array in expression.
- Using Dependency Injection we can replace real objects in production environments with mocked ones for testing environments

IGATE
Speed.Agility.Imagination