Advanced JavaScript

Eng. Niveen Nasr El-Den

iTi

Day 3

Reminder: “Scope” Basics

- A scope is the lifespan of a variable
- Programing languages have block and function scope
- In ES5; only functions have scope. Blocks (if, while, for, switch) do not have scope.
- ES6 presenting let for block scoping
- All variables are within the global scope unless they are defined within a function.
- All global variables actually become properties of the global object (window object).

Reminder: “Scope” Basics

- Variables inside a function are
  - Free var: if they are not declared inside function scope and belong to another scope
  - Bound var: if they are declared inside function
- In JavaScript function scope is lexical/static scope; where free variables belongs to parent scope
- Other language may have dynamic scope where free variables belongs to calling scope.
- JavaScript doesn't have dynamic scope
- When variables are *not declared using the var keyword,* they are declared globally.

Reminder: “Scope” Basics

```
var  myVar = “Hello”; // myVar is a global variable

// create function to modify its own myVar variable
function test (){
      var myVar = “Bye”; //Local variable to test()
       //this is called shadowing
}

test();



alert( myVar); // Global myVar still equals “Hello”
```

Shadowing
occurs when a scope declares a variable that has the same name as one in a surrounding scope; the outer variable is blocked in the inner scope

Reminder: “Scope” Basics

```
var  myVar = “Hello”; // myVar is a global variable

// create function to modify its own myVar variable
function test () {

      /*var*/ myVar = “Bye”;
      //Global myVar’s value has been changed
}

test();

alert( myVar); // Global myVar still equals “Bye”
```

Privileged Method

- The term *privileged method* is not a formal construct, but rather a technique.
- It’s coined by Douglas Crockford
- Privileged methods essentially have one foot in the door:
  - They can access private methods and values within the object
  - They are also publicly accessible

Privileged Method

```
var User = function (name, age) {
        var year = ((new Date().getFullYear() )- age);
        //year is a local variable → private member
        this.getYearBorn = function () { return year;};
};
// Create a new User
var user_1 = new User( “Aly”, 25 );
// Access privileged method to access private year value
alert( user_1.getYearBorn());
alert( user_1.year); // undefined because year is private
```

Private Methods

- Private methods are functions that are only accessible to methods inside the object and cannot be accessed by external code.

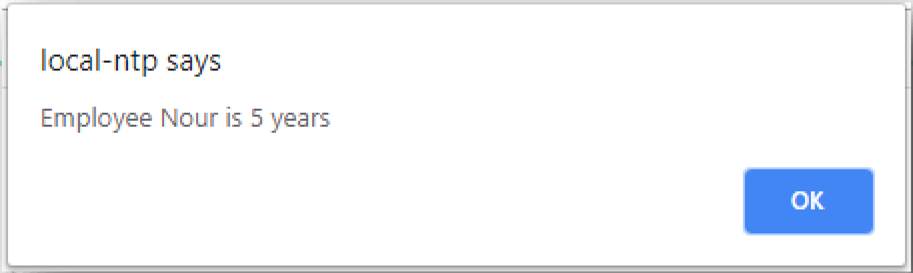
```
var User = function (name) {
      this.name = name;
        function welcome () {
        alert( “Welcome back, ” + this.name + “.”);}
      welcome();
}
// Create a new User
var me = new User( “Aly” ); // alerts: “Welcome back, Aly.”
me.welcome();  // Fails because welcome is not a public method
```

Inner Function = Nested Function

Private Method

“this” and Closure

```
function Employee(name, age){
    this.eName = name;
    this.age = age;
    this.show =  function () {
        var that=this; //_that|_self|_this
        setTimeout(function (){
            alert("Employee " + that.eName + " is " + that.age + " years”);
        },5000);
    }
}
```


local-ntp says

Employee Nour is 5 years

```
var me = new Employee(“Nour”,5);
me.show()
```

✓

“this”, Closure and Private Method

```
function Employee(name, age,yr){
     this.eName = name;
     this.age = age;
     var yrbrn=yr;

    function welcoming() {
      alert("welcome " + this.eName + " you were born in " + yrbrn );
   }

   this.welExec=function(){


      return val();
   }

   //welcoming.call(this);
   var val=welcoming.bind(this) //val();
}
```

Hard binding

no matter what is the invocation context.

Make a function that calls internally and manually an explicit binding and force to do the same instruction

no matter where and how you invoke that function

Class Properties & Methods

- Class Properties and methods are similar to static properties and methods in other object oriented languages.

- This can be created by adding either property or method to a constructor function object.

- This is possible because functions in JavaScript are plain objects that can have properties
  and methods of their own.

Class Properties & Methods

```
function Employee(name, age){
        this.name = name;
        this.age = age;
}

Employee.count=0;


Employee.getCount=function(){
   return Employee.count
}
```

Example!

delete Operator

- The delete operator removes a given property from an object
- If the property which you are trying to delete does not exist, delete will not have any effect and will return true.
- Any var cannot be deleted from the global scope or from a function's scope.
- The delete operator has nothing to do with directly freeing memory
- Memory management is done indirectly via breaking references.

Property Descriptors

Property Descriptors

- Property descriptors hold descriptive information about object properties.

- Property Descriptors allows developer to control some of the internal attributes of the object properties it can be either
  - ▻ Data Descriptor or,
  - ▻ Accessor Descriptor

- To define Property Descriptors use
  Object.defineProperty(obj,”prop”,{})
  Object.defineProperties(obj,{})

Data Descriptor

- A *data descriptor* is a property that has a value, which may be read-only. It is represented by the following keys
  - value : the value associated with the property. Default value is undefined.
  - writable : a Boolean value that determines whether or not the property value can be changed within an assignment operator.
    Default value is false.

Accessor Descriptor

- An *accessor descriptor* is a property described by a getter-setter pair of functions. It is represented by the following keys.
  - ▻ get : A function which serves as a getter for the property.
    Default is undefined.
  - ▻ set : A function which serves as a setter for the property.
    Default is undefined.

Data & Accessor Descriptors Shared Fields

- Both data and accessor descriptors are objects. They share the following optional keys:
    - configurable : determines whether or not a property descriptor can be changed, and the property can be deleted. Default is false.
    - enumerable : determines whether or not the property is enumerated with all of the other members. Default is false.
      i.e. the property will be iterated over when a user does for (var prop in obj){} (or similar).

Descriptors Identifying Fields

Fields	DATA DESCRIPTOR	ACCESSOR DESCRIPTOR	Default Value
value	✓		undefined
writable	✓		false
enumerable	✓	✓	false
configurable	✓	✓	false
get		✓	undefined
set		✓	undefined

Data Descriptors Example

```
var Employee = function(nme, age){
    var person = {};

    Object.defineProperty (person, “nm”, {value : nme, writable :
       true, configurable: true, enumerable: true } );

    Object.defineProperty (person, “age”, {value : age} );

    Object.defineProperty (person, “show”, {value : function (){
                alert("Employee " + this.nm + " is " + this.age
                      + " years old.");
         }
    } );

    return person;

    }
```

Data Descriptors Example

```
var Employee = function(name, age){
        var person = {};

        Object.defineProperties (person,{
                nm:{
                        value : name,
                        writable : false},
                age:{…..},
                show:{…..}
        ……..

        } );

        return person;

    }
```

Accessor Descriptors Example

```
var Employee = function(name, age){
var emp= {};
Object.defineProperty (emp, “nm", {
     get : function() { return name; },
     set : function(val) { name = val; }
      });
return emp;
}
var e= new Employee();
e.nm = “Nour";
var t_emp = e.nm; // alert(t_emp)
```

Example!

value, get & set fields

- An object property cannot have both the *value* and *getter/setter* descriptors. You've got to choose one.

- *Value* can be pretty much anything
  - i.e. primitives or built-in types or even be a function.

- You can use the *getter* and *setters* to mock read-only properties.

- You can even have the *setter* throw Exceptions when users try to set it.

Reminder : Object Object Properties & Methods

- .hasOwnProperty(“prop”)
- .valueOf()
- .toString()

- Object.keys(obj) → enumerable properties
- Object.getOwnPropertyNames(obj) → enumerable and non-enumerable properties

- Object.defineProperty(obj,”prop”,{})
- Object.defineProperties(obj,{})

- Object.getOwnPropertyDescriptor(obj,prop)
- Object.getOwnPropertyDescriptors(ctor.prototype)

- Object.create(obj [,{}])
- …

Other Useful Object Methods

- Object.seal()
  - ▻ Marks every existing property on the object as *non-configurable*
  - ▻ Then call *Object.preventExtensions* to prevent adding new properties

- Object.freeze()
  - ▻ Mark every existing property on the object as non-writable
  - ▻ Invokes *Object.seal* to prevent adding new properties and marks existing properties
    as non-configurable

in Operator
vs
.hasOwnProperty()

Prototype Property

Prototype Property

- **Prototype:** is a property that allows you to add more properties and methods to any created object.
- It is a property of the function objects that gets created as soon as you define a function.
- Attaching new properties to a prototype will make them a part of every object instantiated from the original prototype, effectively making all the properties public (and accessible by all).
- This is another way to add more functionality to already created objects using constructor function
- It is also used for inheritance

Prototype Property & Public Method

- Public methods are completely accessible by the end user.
- Public method is a property of the function objects
- To achieve these public methods, which are available on every instance of a particular object, we need to the *prototype* property

Prototype Property & Public Method

```
function User( name, age ){
    this.name = name;
    this.age = age;
}
```

```
var User = function (name,age){
    this.name = name;
    this.age = age;
}
```

```
// Add a public accessory method for name
User.prototype.getName = function(){
return this.name;};

// Add a public accessory method for age
User.prototype.getAge = function(){
return this.age; };

User.prototype.job="Engineer";
```

Pseudo classical pattern

Prototype Property & Public Method

```
// Instantiate a new User object

var user = new User( “Ahmed", 25 );

alert( user.getName()); //Ahmed

alert( user.getAge()); //25

alert(user.job); //Engineer
```

Example!

Overriding

occurs when two methods having the same method name and parameters (i.e., *method signature*) where one of the methods is implemented in the parent class while the other is implemented in the child class, so that a child class provides a specific implementation of a method that is already provided its parent class.

Prototype Property & Overriding Methods

- override methods when its required to be different from the available property

```
// overridding toString() for User object
User.prototype.toString = function(){
return “user name is: ”+this.name+”and his age is: “+this.age;};

document.write(user.toString().);
// user name is: Ahmed and his age is: 25
```

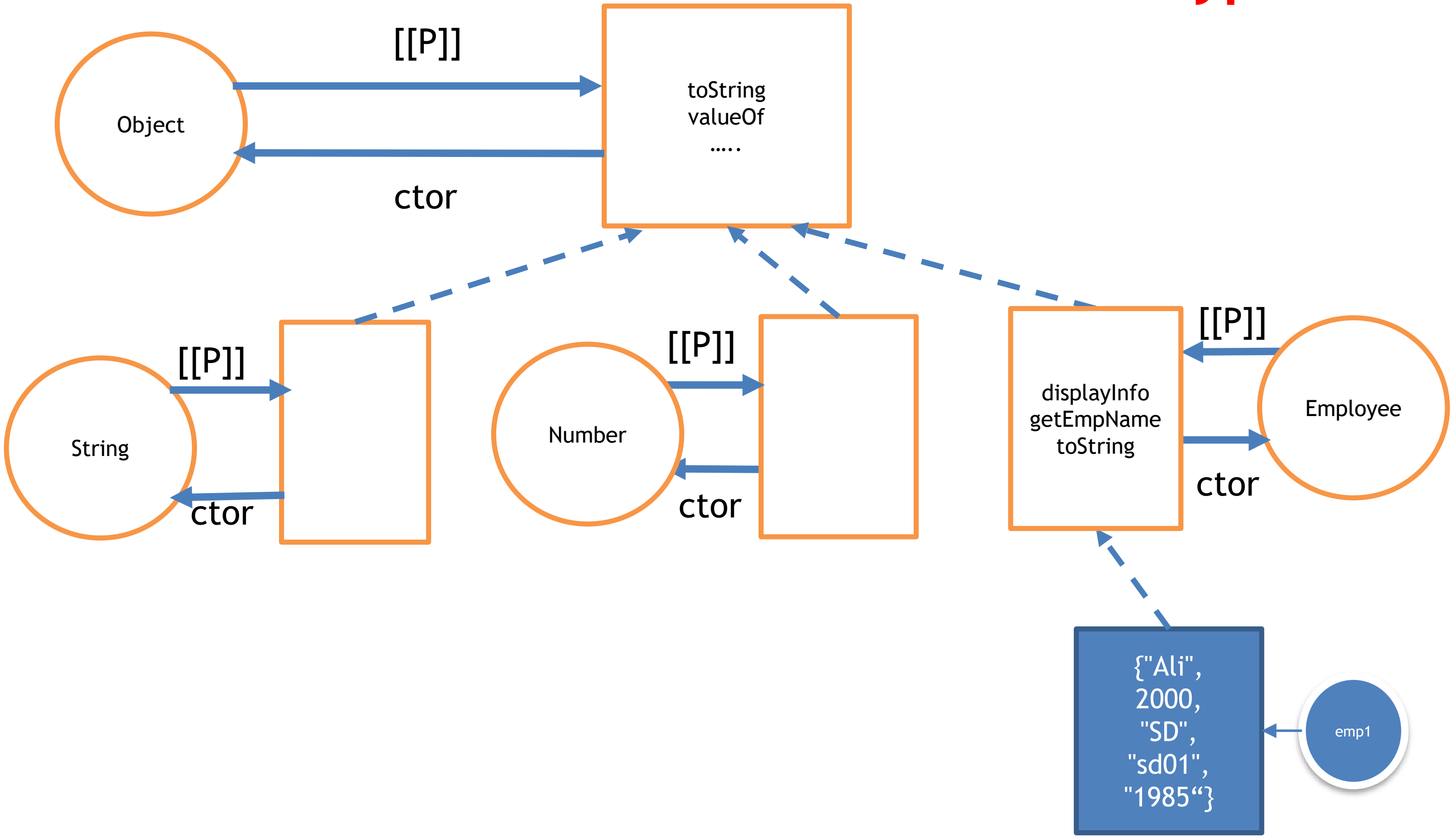
Prototype Chaining

- Prototype chaining is used to build new types of objects based on existing ones. It has a very similar job to inheritance in a class based language

- A mechanism for making objects that resemble another object when we want these object to have same properties

- Make one object behave as if it has all of the properties of another object by delegating a lookups from the 1st to the 2nd

Prototype Chaining



var emp1= new Employee(“ali”,200,”SD”,”sd01”,”1985”)

In

Prototype Chain Mechanism

if an object does not know how to retrieve a property,

it tries to ask the object above in the chain.

It **delegates**.

Assignment