Advanced JavaScript

Eng. Niveen Nasr El-Den
iTi

Day 2

These are the Golden Days of JavaScript

Inner Functions

- Functions can be defined within one another
- Inner functions have access to the outer function's variables and parameters.

```
function getRandomInt(max) {
var randNum = Math.random() * max;
        function ceil() {
        return Math.ceil(randNum);
        }
return ceil(); // Notice that no arguments are passed
}

// Alert random number between 1 and 5
alert(getRandomInt(5));
```

Inner Function; only accessible within getRandomInt

Inner Functions

```
function a(param) {
    function b(theinput)    {
        return theinput * 2;
    };

    return 'The result is ' +
        b(param);
};
```

```
var a = function(param) {
    var b = function(theinput) {
        return theinput * 2;
    };

    return 'The result is ' +
        b(param);
};
```

- a(2); →"The result is 4“
- a(8); →"The result is 16“
- b(2); →b is not defined

Example!

Inner Functions

- The nested (inner) function is private to its containing (outer) function.
- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure:
  - The inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function
    - i.e. The inner function contains the scope of the outer function.
  - When two arguments or variables in the scopes of a closure have the same name, there is a name conflict. More inner scopes take precedence
  - According to scope chain, the inner-most scope is the first on the chain.

Inner Functions

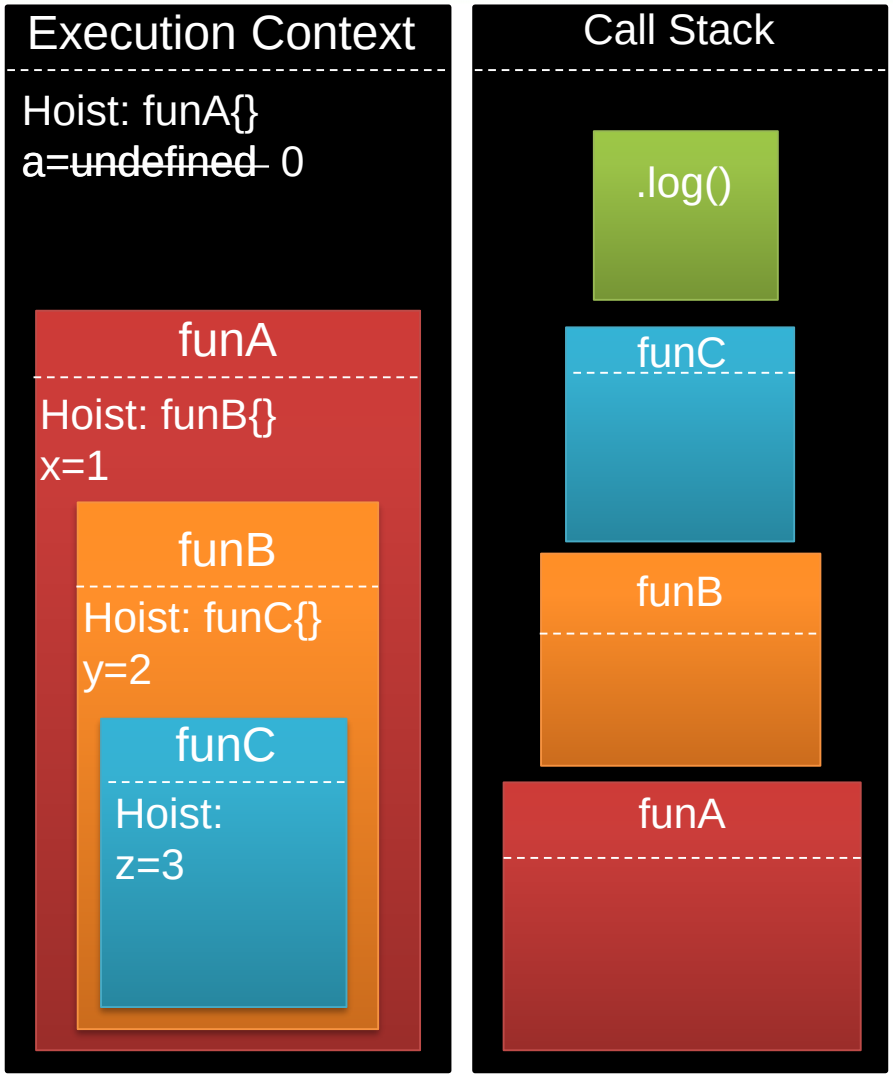
```
function myFun(x) {
    var z = 10;

    function innerFun(y) {
        return x + y + z;
    }

    return innerFun;
}
```

```
var myFun = function (x) {
    var z = 10;

    return function (y) {
        return x + y + z;
    };

}
```

```
var fun = myFun(5);
var result = fun(10);
```

```
var result = myFun(5)(10);
```

Inner Functions




```
function funA(x) {

    function funB(y) {

        function funC(z) {
            console.log(x + y + z + a);
        }

        funC(3);
    }

    funB(2);
}
var a=0;
```

```
funA(1);        //6
```

Inner Functions

```
function fHello(){return “Hello  ”}
var fHey=function(){return “Hey…”}
var x1= fHello();
function fHii(){return “Hii!!!”};

var myFun=function() {
      var x2 = fHey ();

    return function(){
          var x4 =fHii();
          console.log(x1+x4+x2)
    };

}

var fun=myFun();
fun();
```

Hello Hii!!!Hey…

Closures

- Closure is one of the most powerful features of JavaScript.

  “A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).”

  - When function returns from another function it returns with all variables from its external scope
  - A new closure is created for each call to outside.
  - Closure wrap-up the entire environment with all variables from external scope

Closures

- Closure provides a sort of security for the variables of the inner function, since they are not accessed by their outer function
  - This provides a sort of encapsulation for the variables of the inner function
- It is created when the inner function is somehow made available to any scope outside the outer function.
- If the inner function manages to survive beyond the life of the outer function; the variables and functions defined in the outer function will live longer than the outer function itself, since the inner function has access to the scope of the outer function.
- Closure makes currying possible in JavaScript.
- A closure problem occurs inside loops

Closure is
created by the nested functions
to retain access to each of the
arguments.
So inner function have access
to all arguments of
outer scope

Problem

```
function closureTest(){
      var arr = [];

      for(var i = 0; i < 3; i ++) {
            arr.push(function(){
           console.log(i);
             });
      }

      return arr;
}

var cFn = closureTest();

cFn[0]();
cFn[1]();
cFn[2]();
```

Solution

```
function closureTest(){
    var arr = [];

    for(var i = 0; i < 3; i ++) {
        arr.push((function(j) {
                    return function(){console.log(j);}
                })(i)
        );
    }

    return arr;
}

var cFn = closureTest();

cFn[0]();
cFn[1]();
cFn[2]();
```

IIFE Pattern

- A common often extra ordinary used pattern
- Besides advantages and disadvantages of anonymous function, IIFEs are
  - Suitable for initialization tasks
  - Work done without creating global variable
  - Its where the magical part happens in avoiding closures
  - Also, cant execute twice unless it is put inside loop or another function
  - Introduces a new scope that restrict the lifetime of a variable

Error Handling

JavaScript Error Handling

- **There are two ways of catching errors in a Web page:**

  1.try...catch **statement.**

  2.onerror **event.**

try...catch Statement

- The try...catch statement allows you to test a block of code for errors.
- The try *block* contains the code to be run.
- The catch *block* contains the code to be executed if an error occurs.
- Syntax

```
try {
   //Run some code here
}
catch(err) {
   //Handle errors here
}
```

Implicitly an Error object "err" is created

If an exception happens in "scheduled" code, like in setTimeout, then try..catch won't catch It

try...catch Statement (no error)

try {

✓no error.

✓no error.

✓no error.

}

catch(exception)

{

~~✓error handling code will not run.~~

}

✓ execution will be continued.

try...catch Statement (error in try)

try {

- ✓ no error.
- ✓ no error.

an error! *control is passed to the catch block here.*

this will never execute.

}

catch(exception)

{

- ✓ error handling code is run here

}

✓ execution continues from here.

Example!

try...catch Statement (error in catch)

try {

✓ no error.

✓ no error.

an error! *control is passed to the catch block here.*

this will never execute.

}

catch(exception)

{

✓ error handling code is run here

an error!

~~error handling code is run here will never execute.~~

}

~~execution wont be continued.~~

Example!

try...catch & throw Example

```
try{
        if(x<100)
                throw “less100"
        else if(x>200)
                throw “more200"
    }
catch(er){
                if(er==“less100")
                    alert("Error! The value is too low")
                if(er == “more200")
                    alert("Error! The value is too high")
            }
```

Example!

Adding the *finally* statement

- **If you have any functionality that needs to be processed regardless of success or failure, you can include this in the *finally* block.**

try…catch…finally Statement (no error)

try {

- ✓ no error.
- ✓ no error.
- ✓ no error.

}

catch(exception)

{

- ✓ ~~error handling code will not run.~~

}

finally {

- ✓ This code will run even there is no failure occurrence.

}

- ✓ execution will be continued.

try…catch…finally Statement (error in try)

try {

- ✓ no error.
- ✓ no error.

an error! *control is passed to the catch block here.*

this will never execute.

}

catch(exception)

{

- ✓ error handling code is run here
- ✓ error handling code is run here
- ✓ error handling code is run here

}

finally {

- ✓ This code will run even there is failure occurrence.

}

✓ execution will be continued.

Example!

try…catch…finally Statement (error in catch)

try {

- ✓ no error.
- ✓ no error.

an error! *control is passed to the catch block here.*

this will never execute.

}

catch(exception)

{

- ✓ error handling code is run here

an error!

~~error handling code is run here will never execute.~~

}

finally {

- ✓ This code will run even there is failure occurrence.

}

~~execution wont be continued.~~

Example!

onerror Event

- The old standard solution to catch errors in a web page.

- The *onerror* event is fired whenever there is a script error in the page.

- onerror event can be used to:
  - Suppress error.
  - Retrieve additional information about the error.

Suppress error

```
function supError()  {
        alert("Error occured")
  }
  window.onerror=supError
```

OR

```
  function supError()    {
        return true; //or false;
  }

  window.onerror=supError
```

The value returned determines whether the browser displays a standard error message.

true the browser does not display the standard error message.

false the browser displays the standard error message in the JavaScript console

Retrieve additional information about the error

```
onerror=handleErr

  function handleErr(msg,url,l,col,err) {
        //Handle the error here
        return true; //or false;
  }
```

where

- msg → Contains the message explaining why the error occurred.
- url → Contains the url of the page with the error script
- l → Contains the line number where the error occurred
- col → Column number for the line where the error occurred
- err → Contains the error object

JavaScript is designed on a simple **object-based paradigm**

JavaScript is
Multi-paradigm
Programming Language.

JavaScript supports programming in many different styles.

Object-Oriented JavaScript

Object-Oriented JavaScript

- The main principle with OOP is the use of Classes to create objects, and that objects are implemented in a manner that allows them to adopt Inheritance, Polymorphism, and Encapsulation.

- In most other object-oriented languages you would instantiate an instance of a particular class, but that is not the case in JavaScript.

- Unlike most other object-oriented languages, JavaScript doesn't actually have a concept of classes. It looks and behaves differently.

Object-Oriented JavaScript

- JavaScript is a *class-free*, object-oriented language
- Although ES6 introduces JavaScript class expressions and class declarations, to provide a much clearer syntax to create objects and deal with.
- In fact classes are functions
- Custom Object that you, as a JavaScript developer, create and use is the main actor in application.

Custom Object

- Objects that you, as a JavaScript developer, create and use.

- An object in JavaScript is a complex construct usually consisting of a constructor as well as zero or more methods and/or properties.

- Objects can be either stand-alone with their own set of properties & functions or they can inherit properties from other objects

Custom Object

- There are different ways to create an instance of an object class (Functions in JavaScript)
  - Basic Object Literal Pattern
  - Factory Function
  - Custom Object Constructor Function
  - …

Literal Pattern Object Creation

```
var obj =  { };

obj.name = “banana”

obj.click = function(){
    alert( “you can eat" );
    }

obj.details ={
        mycolor: ”yellow”,
        mycount:12
        }

//(obj instanceof Object) // true
```

Literal Pattern Object Creation

```
var obj = {
// Set the property names and values use key/value pairs
    “name” : “banana”, // name : “banana”
    click : function(){
        alert( “you can eat" );
        },
    //initialize entire object
    details : {
            mycolor: ”yellow”,
            mycount:12
          }
};
```

Custom Object creation using basic object literal pattern

- **We can create objects with a short syntax that defines an object inside curly braces. (basic object literal pattern)**

```
var emp1 = { name:“Aly", age: 23};

var emp2 = { name: “Hassan",  age: 32};
```

Custom Object creation using basic object literal pattern

- After an object exists, you can add a new property to that instance by simply assigning a value to the property name of your choice.

- For example, to add a property about the “Salary” for “Hassan”, the statement is:

```
var emp1 = { name:“Aly", age: 23);
var emp2 = { name: “Hassan",  age: 32);
emp2.salary = 320;
```

- After that assignment, only emp2 has that property.

- There is no requirement that a property be pre-declared in its constructor or shortcut creation code.

Custom Object creation using basic object literal pattern

- After an object exists, you can add a new property to that instance by simply assigning a value to the property name of your choice.

- For example, to add a property about the “Salary” for “Hassan”, the statement is:

```
var emp1 = { name:“Aly", age: 23);
var emp2 = { name: “Hassan",  age: 32);
emp2.salary = 320;
```

- After that assignment, only emp2 has that property.

- There is no requirement that a property be pre-declared in its constructor or shortcut creation code.

Factory Function Pattern

- It is a way where object is created as a return of a function call assigned to a variable
- Used to create Multiple Objects with same interface
- No need to use “new” when calling a factory function

Creating New Instance from Custom Object using Factory Pattern

- Factory Function for Employee Object

```
var Employee = function (e_nm, e_ag){
    return {
        name : e_nm,
        age : e_ag
        }
}
```

```
var Employee = function (e_nm, e_ag) {
    var emp = { name : e_nm,
                age : e_ag
              };
    return emp;
}
```

- Creating object instances using Factory Function Method

```
var emp1 = Employee (“Aly", 23);

var emp2 = Employee (“Hassan", 32);

var emp3 = Employee ();
```

Constructor Function

- A constructor function looks like any other JavaScript function, but its purpose is:
  - to define the initial structure of an object
  - to define it's property and method names
  - It can populate some or all of the properties with initial values.
  - Values to be assigned to properties of the object are typically passed as parameters to the function,
  - Statements in the constructor function assign those values to properties.
- MyConstructor
- myFunction

Creating New Instance from Custom Object using Constructor Method

- Constructor Function for Employee Object

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

- To create object instances using Constructor Function Method, invoke the function with the new keyword

```
var emp1 = new Employee (“Aly", 23);

var emp2 = new Employee (“Hassan", 32);

var emp3 = new Employee ();
```

Adding methods to Constructor Function (Functional shared Pattern)

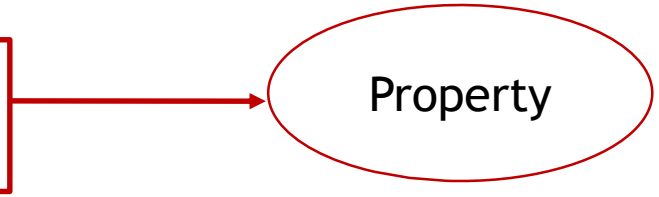
- Functional shared pattern is used to save memory by adding methods to the constructor function:

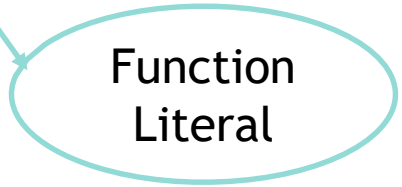
```
function Employee(name, age){
    this.name = name;
    this.age = age;
    this.show = showAll;
}

function showAll( ){
    alert("Employee " + this.name + " is " + this.age + " years old.");
}
```

Adding methods to Constructor Function (Functional Class Pattern)

- Adding methods to the constructor function using Function Literal:

```
function Employee(name, age) {
    this.name = name;
    this.age = age;
    this.show =  function () {
        alert("Employee " + this.name + " is " + this.age + " years old.");
    }
}
```





Instance Object Creation

```
// Class using constructor function
function User( name ) {
    this.name = name;
        this.display = function(){return this.name;}
}

// Instance object of user
var me = new User( "My Name" );

// Test
alert( me.name );
alert( me.display());
alert( me.constructor == User);//true
alert( me.constructor == Object);// false
```

Reminder:
Function Default arguments

```
function myFun(){
	var x = arguments[0] || 10;
	var y = arguments[1] == undefined ? 11 : arguments[1]

	return x + y;
}


myFun(); //21
myFun(1); //12
myFun(1,2); //3
```

Creating New Instance from Custom Object via Constructor Overloading

- Assign a *default value* to a Property:

```
function Employee (id='idx' /*ES6*/),name, age,salary=2000/*ES6*/){
    this.name = typeof name ==="undefined"? "Nour": name;
    this.age = age|| 0; //ES5
    this.salary = salary;
    this.id = id;
}
```

- We can also generate a blank object and then populate it explicitly; property by property:

```
var emp1 = new Employee( );
emp1.name = "Aly";
emp1.age = 23;
```

Overloading

- A common feature in other object-oriented languages is the ability to “overload” functions.
- Overloading occurs when more than one method within the same class have the same method name but different in parameters (different numbers and/or types of passed arguments) to perform different behaviors
- Overloading can be fulfilled via
  - function arguments property using default parameters & any conditional statement.
  - Creating function that calls the meant function with proper requirement

Constructor Function, new & this

- When function invoked with new, functions return an object known as this.
- “new” before any function call turns it into constructor function
- JavaScript uses “this” keyword to refer to the current object.
- “this” is confusing sometimes, when it doesn’t return the expected object.
- You have a chance of modifying this before it is returned

“new” Operator

- When using “new”
  - A brand new empty object is created
  - That object get linked to another object
  - It gets bound as “this” keyword as a purpose for function call
  - If the function doesn’t return any thing, it will return “this”.
- Note:
  - Primitive datatypes pass by value while Objects and Arrays pass by reference;
    - When any change happens in obj1 it is reflected in obj2
  - using new can over come this problem

With the exception of null and undefined, all primitives values have object equivalents which wrap around the primitive values.

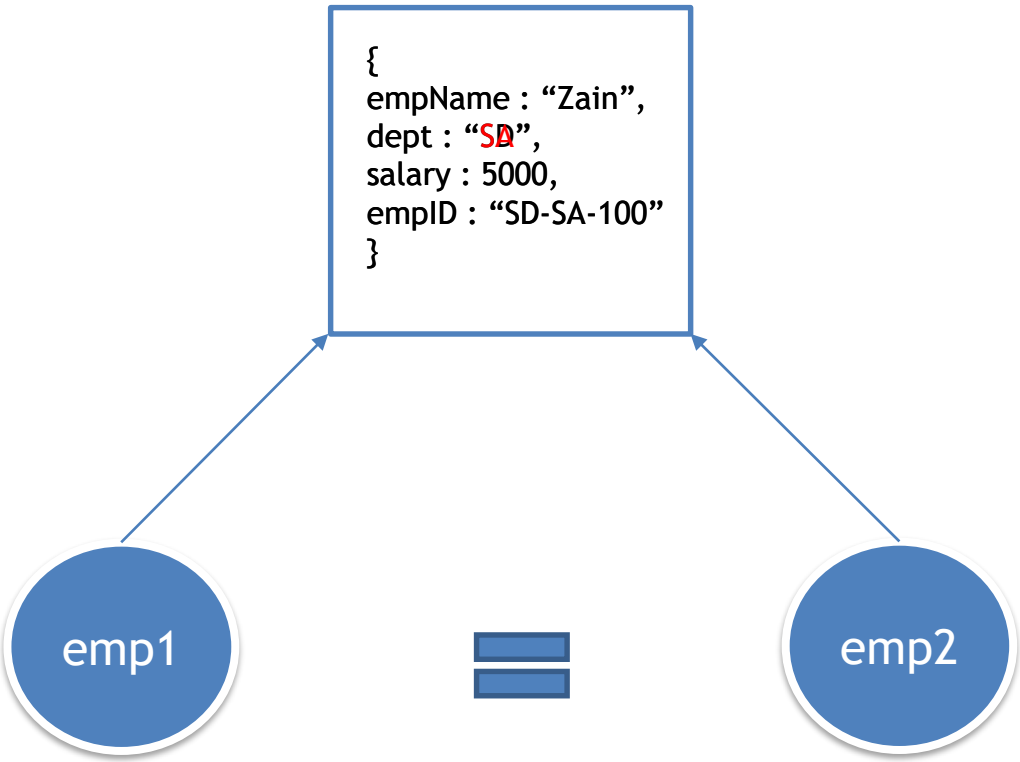
All primitives are **immutable**

All objects are **reference** values

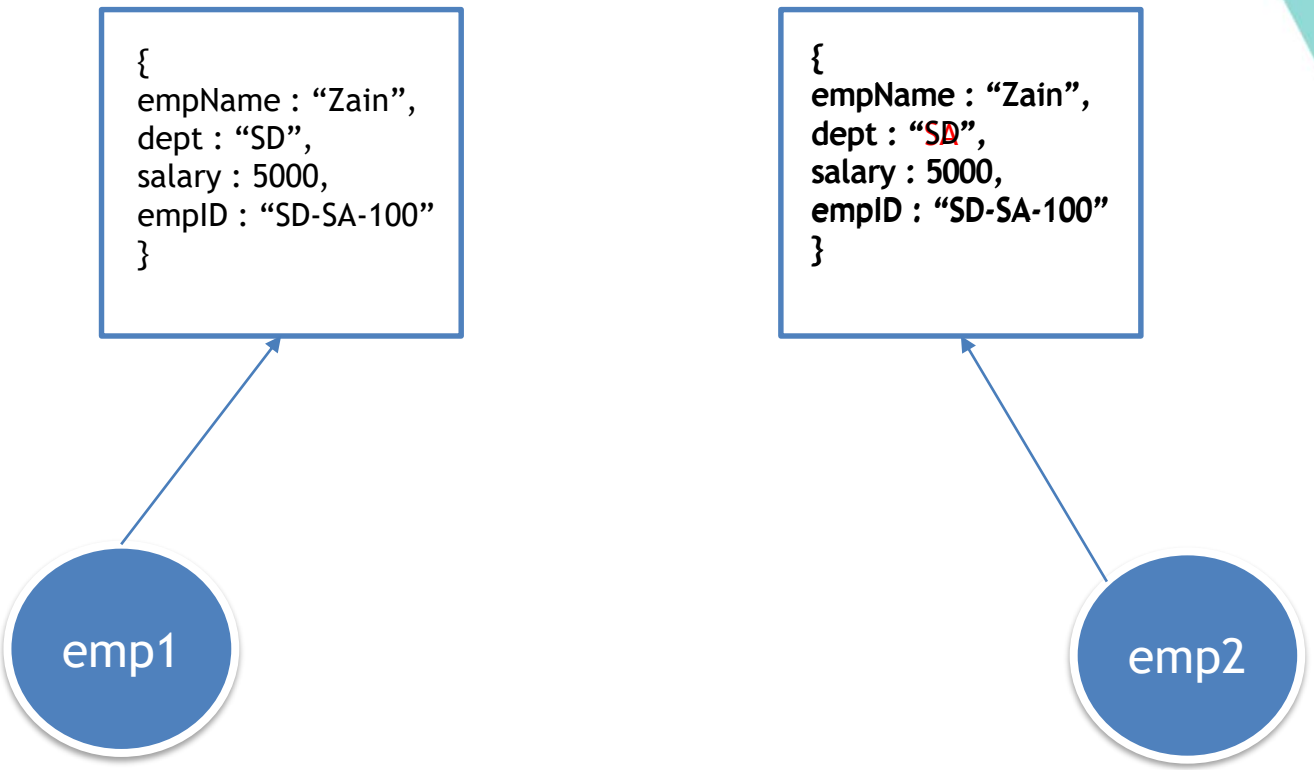
References

- Reference is a pointer to an actual location of an object
- An object can contain a set of properties, all of which are simply references to other objects.
- When multiple variables point to the same object, modifying the underlying type of that object will be reflected in all variables

Example!




```
var emp1 = new Employee(“Zain", 5000, "SD“, “SD-SA-100”);

var emp2 = emp1;

emp2.dept = “SA”;

console.log(emp1.dept);//SA
```




```
var emp1 = new Employee(“Zain", 5000, "SD“, “SD-SA-100”);

var emp2 = new Employee();

for( var i in emp1)
        emp2[i] = emp1[i];

emp2.dept = “SA”
console.log(emp1.dept);//SD
console.log(emp2.dept);//SA
```

“this” keyword & Binding

- Every function while executing, has a reference to its execution context called “this”.
- “this” is an identifier that gets the value of object bound to it, it behaves like normal parameters.
- “this” binding is dependent on its “call site” (where the function get executed)

“this” keyword

- “this” is dynamic since it looks for things at runtime, based upon how you call things

- 4 rules for binding “this” (in terms of order precedence) depending on call site
  - Default Binding
  - Implicit Binding
  - Explicit Binding
  - new keyword

Hard Binding

Default & Implicit Binding

- Default Binding
  - It is applied on a standalone functions & IIFEs
    - Function defined in Global Scope
  - Depends on Strict Mode of code running inside a function
    - Its value is undefined in strict mode,
      - To be applied globally should be called as window.fn
    - otherwise its value is Global Object
- Implicit Binding
  - An object is calling the function
    - Object on the left of the (.) function call

Example 1

```
function myFun(){
      console.log(this.val)
}
var val = “myVal”;

var myObj1 = {val : ”obj1Val”, myFun: myFun};

var myObj2 = {val : ”obj2Val”, myFun : myFun};

myFun(); //myVal
myObj1.myFun(); //obj1Val
myObj2.myFun(); //obj2Val
```

```
function myFun(){
        var val= "myVal";
        this.val= "myNewVal"
         this.fun=fun;
         this.fun();
         fun();
}

function fun(){
         console.log(this.val)
}

var val = "globalVal";

myFun();  //new myFun();
console.log(val);//???
```

Explicit Binding

- It’s a hard binding
- When function is called, it predict its object
- If you want to set a specific object other than the calling object make hard binding using Function Object methods.
  - bind()
  - apply()
  - call()

Example!

Using call() and apply()

```
var myObj={
      name:"myObj Object",
      myFunc:function(){
                  alert(this.name)
            },
      myFuncArgs:function(x,y){
                  alert(this.name+" " + x +" "+y)
            }
};

var obj1={name:"obj1 Object"};
```

```
myObj.myFuncArgs(1,2);      //myObj Object 1 2

myObj.myFuncArgs.apply(obj1,[1,2]);    //obj1 Object 1 2

myObj.myFuncArgs.call(obj1,1,2);      //obj1 Object 1 2
```

Using bind()

```
var myObj={
        name:"myObj Object”,
        myFunc:function(){
                        alert(this.name)
                },
        myFuncArgs:function(x,y){
                        alert(this.name+" " + x +" "+y)
                }
};

var obj1={name:"obj1 Object"};
```

```
myObj.myFuncArgs(1,2);  //myObj Object 1 2

myObj.myFuncArgs.bind(obj1)(5,6); //obj1 Object 5 6
myObj.myFuncArgs.bind(obj1,5)(6); //obj1 Object 5 6
myObj.myFuncArgs.bind(obj1,5,6)(); //obj1 Object 5 6
```

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

Assignment