Advanced JavaScript

Eng. Niveen Nasr El-Den
iTi

Day 4

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

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

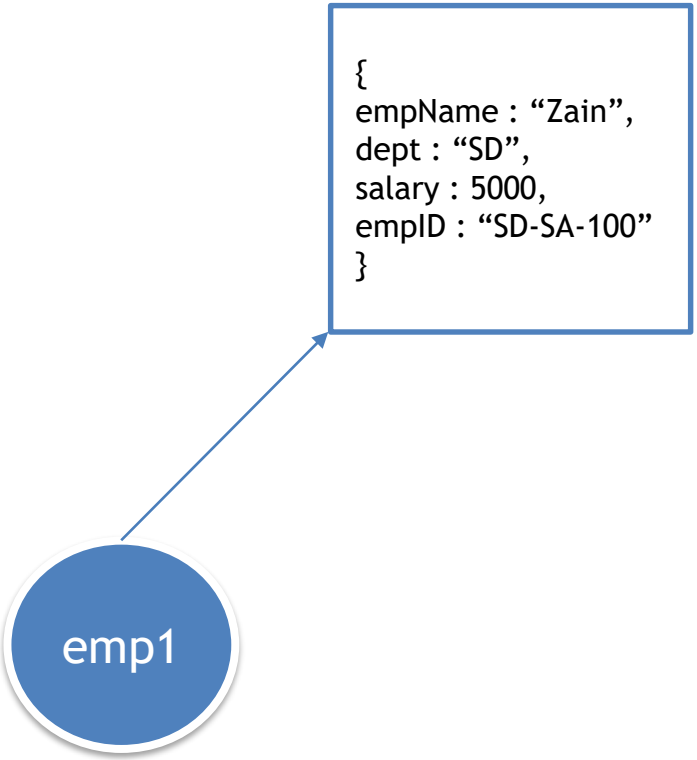
All primitives are immutable

All objects are reference values

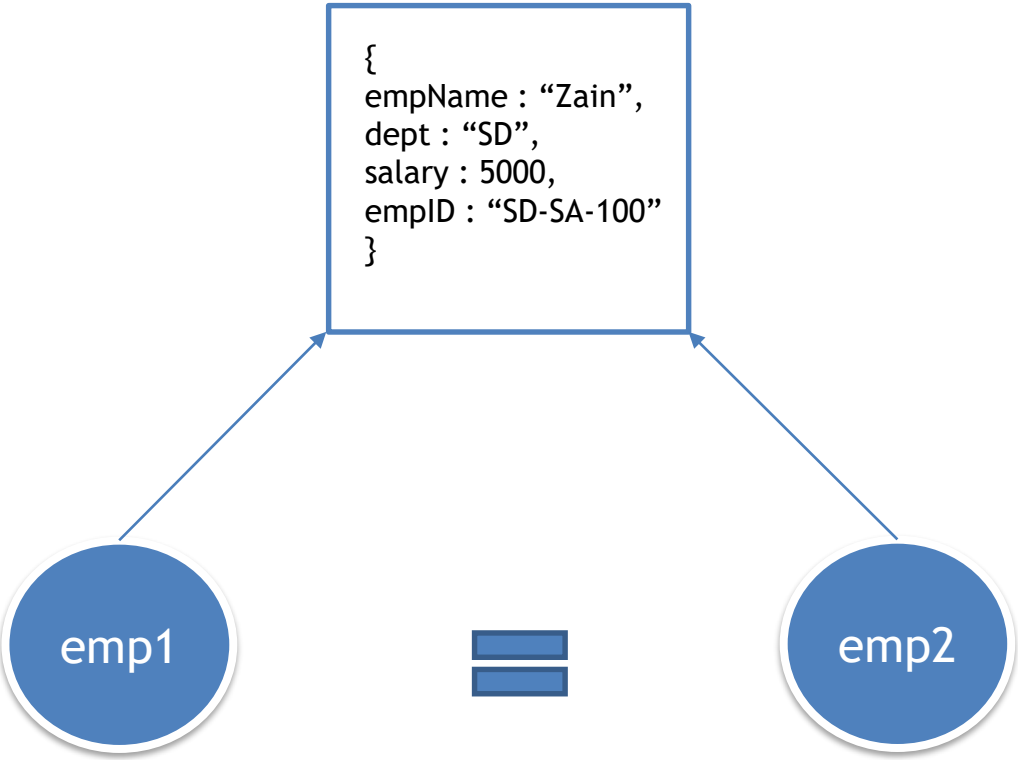
References

- 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

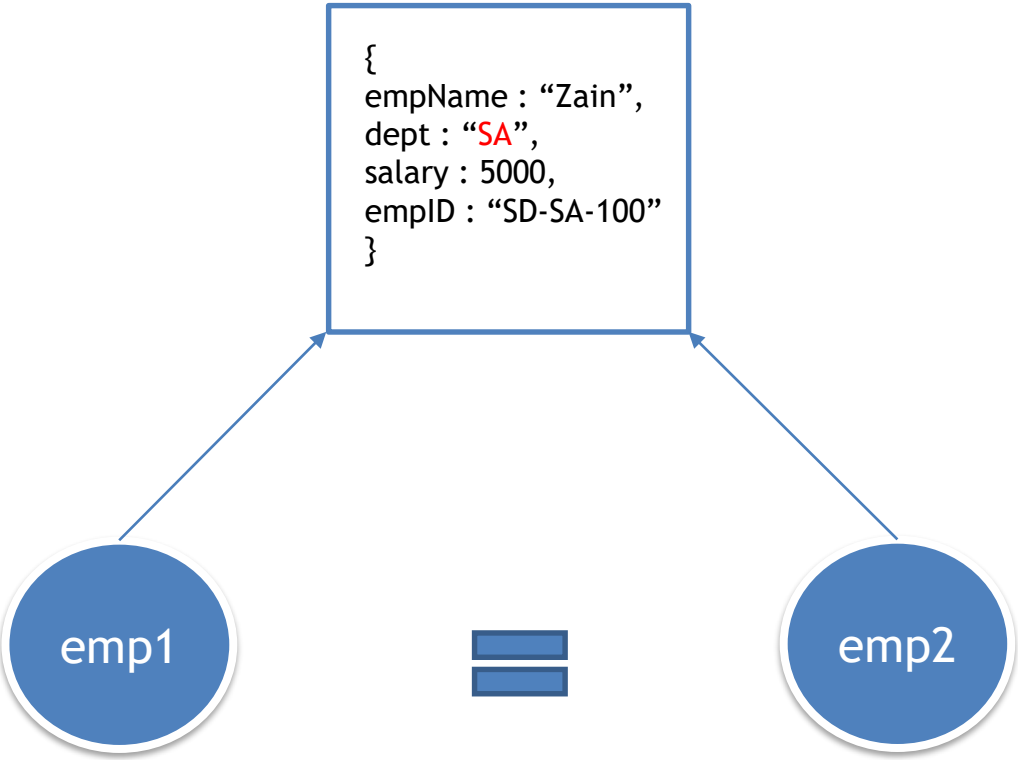
Example!



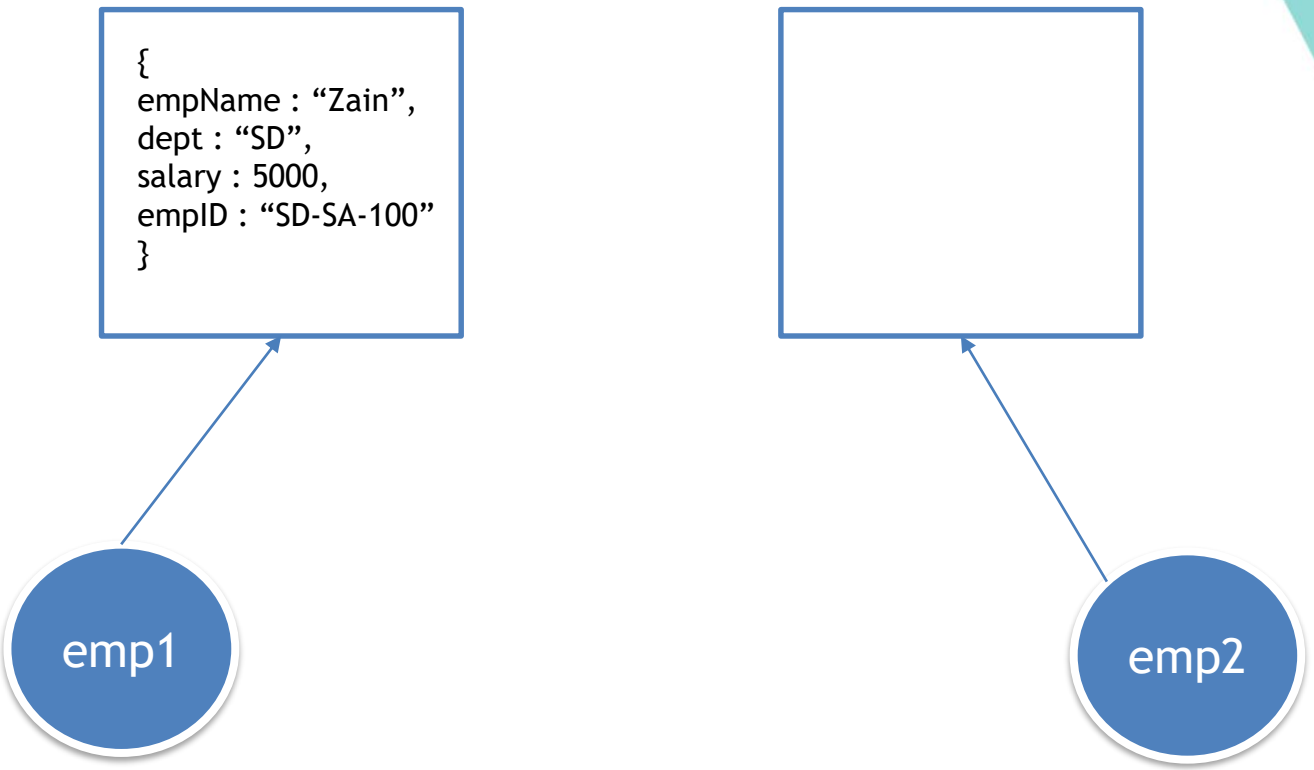
var emp1 = new Employee(“Zain", 5000, "SD“, “SD-SA-100”);




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

var emp2 = emp1;
```

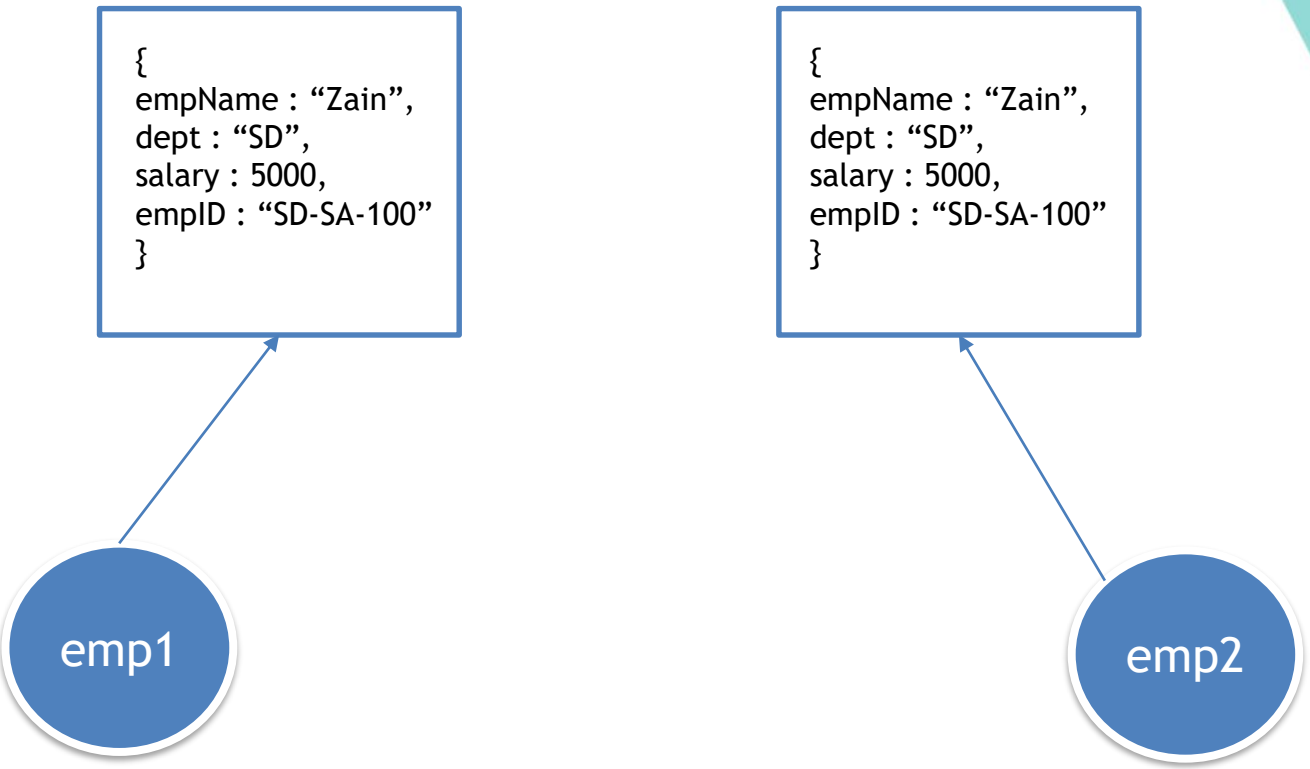



```
var emp1 = new Employee(“Zain", 5000, "SD“, “SD-SA-100”);

var emp2 = emp1;

emp2.dept = “SA”;

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

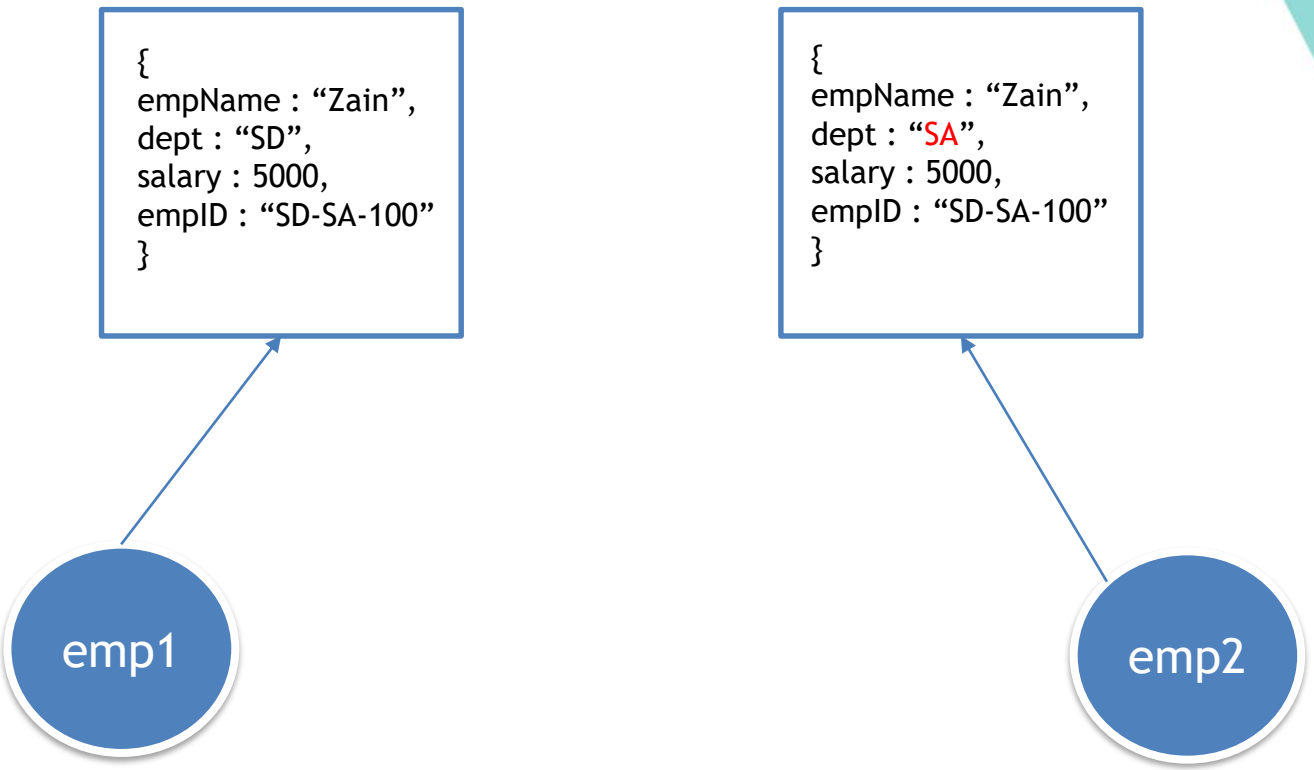


var emp1 = new Employee(“Zain", 5000, "SD“, “SD-SA-100”);

var emp2 = new Employee();




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

var emp2 = new Employee();

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




```
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
```

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(nm, age){
        var person = {};

        Object.defineProperties (person,{
                nm:{
                        value : nm,
                        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
```

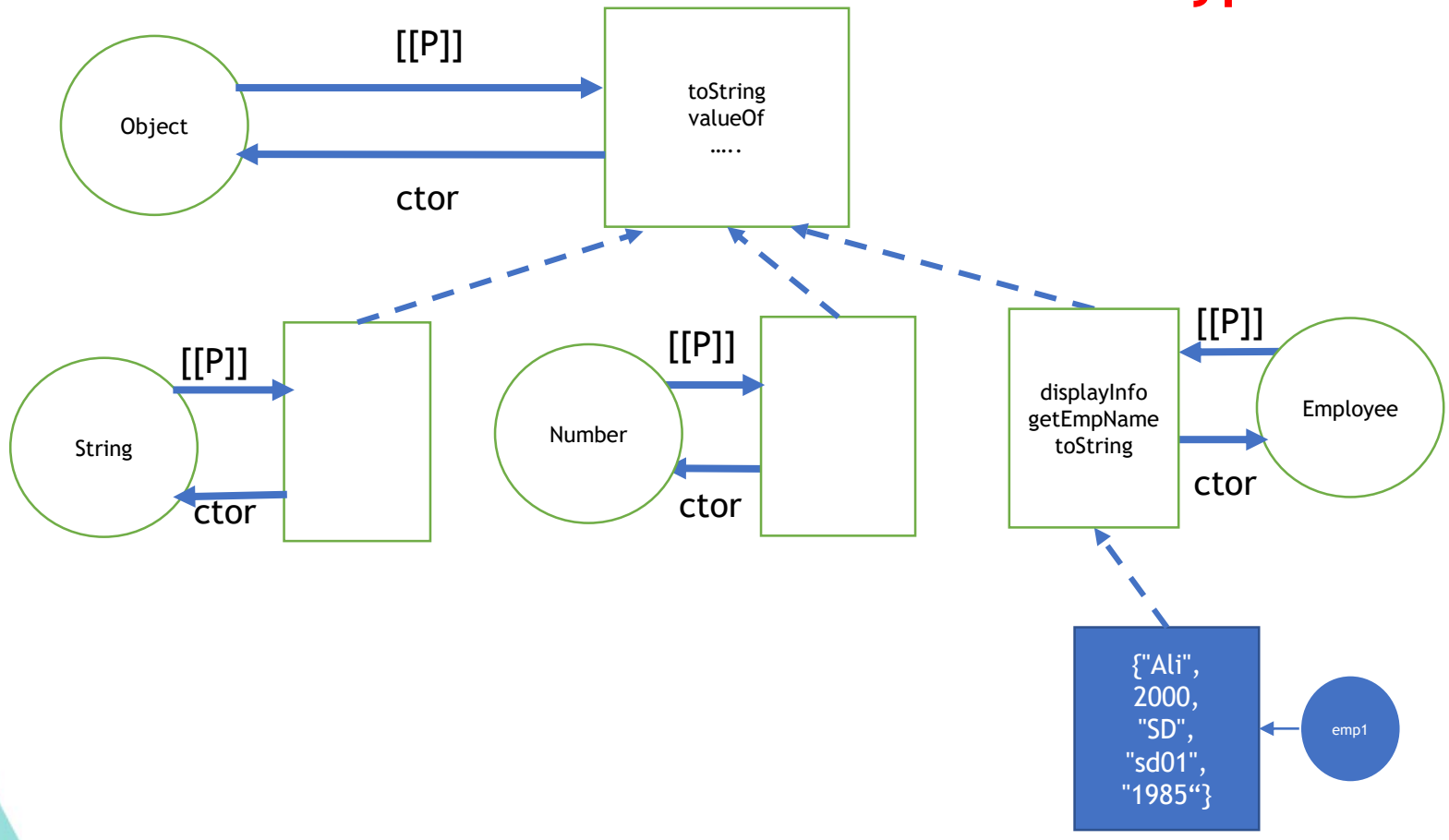
Prototype Chaining

- 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”)

Inheritance

JavaScript is a
prototype based language

It is also known as
instance-based programming,
prototype-oriented,
prototypal, or
classless.

https://en.wikipedia.org/wiki/Prototype-based_programming

prototype based language
is a style of
object-oriented programming
in which behavior reuse
(known as **inheritance**)
is performed via a process
of reusing existing **objects**
that serve as
prototypes

https://en.wikipedia.org/wiki/Prototype-based_programming

Inheritance

- Inheritance refers to an object being able to inherit methods and properties from a parent object (a Class in other OOP languages, or a Function in JavaScript).

- There are different ways for a child to inherit from a parent
  - Parasitic Inheritance
  - Prototypal Inheritance
  - etc..

Inheritance in JavaScript works through something called prototypes and this form of inheritance is often called prototypal inheritance

Prototypal Inheritance

- JavaScript uses a unique form of object creation and inheritance called *prototypal inheritance*

- It is the default way to implement inheritance, and is described in the ECMAScript standard.

- The premise behind this is that an object constructor can inherit methods from one other object, creating a *prototype object from which all other new objects are built.*

Prototypal Inheritance

- This process is facilitated by the *prototype property.*
- Prototypes do not inherit their properties from other prototypes or other constructors; they inherit them from physical objects.
- Prototype chaining (pseudo-classical)

```
Child.prototype = new Parent();
```

- Inherit only the prototype

```
Child.prototype = Parent.prototype;
```

Prototypal Inheritance

```
// Using prototype object for inheritance behavior
        function Employee(name, age){
            this.name = name;
            this.age = age;
        }

// No Classes, use constructor functions
        function Person(name){
            this.name = name;
        }
// Public functions using prototype
        Person.prototype.getName = function() {
            return this.name;
        }
```

Prototypal Inheritance

```
// Inherit only the prototype
    Employee.prototype = Person.prototype;

   var e = new Employee('Ahmed', 12);
   e.getName();    //Inherited from Person
```

```
// prototype chaining
    Employee.prototype = new Person();

   var e = new Employee('Ahmed', 12);
   e.getName();    //Inherited from Person
```

Parasitic Inheritance

- Parasitic inheritance was first proposed as a way of handling inheritance in JavaScript by Douglas Crockford in 2005.

- In this methodology you create an object by using a function that copies another object, augments it with the additional properties and methods you want the new object to have and then returns the new object.

```
var childObjProto=Object.create (parentObjProto)
```

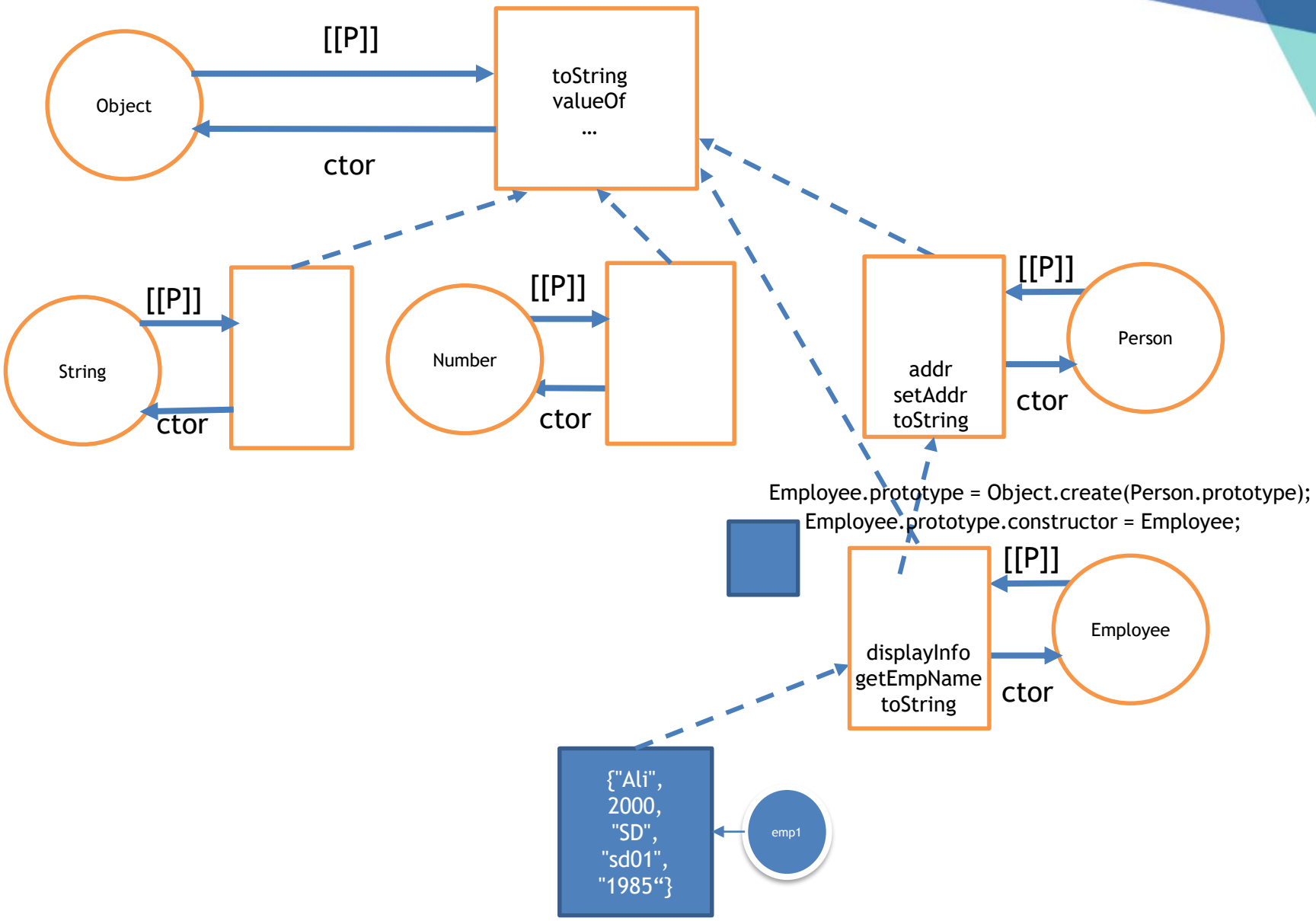
Parasitic Inheritance

- using create() creates an empty object , its proto is pointing to the object passed as a parameter

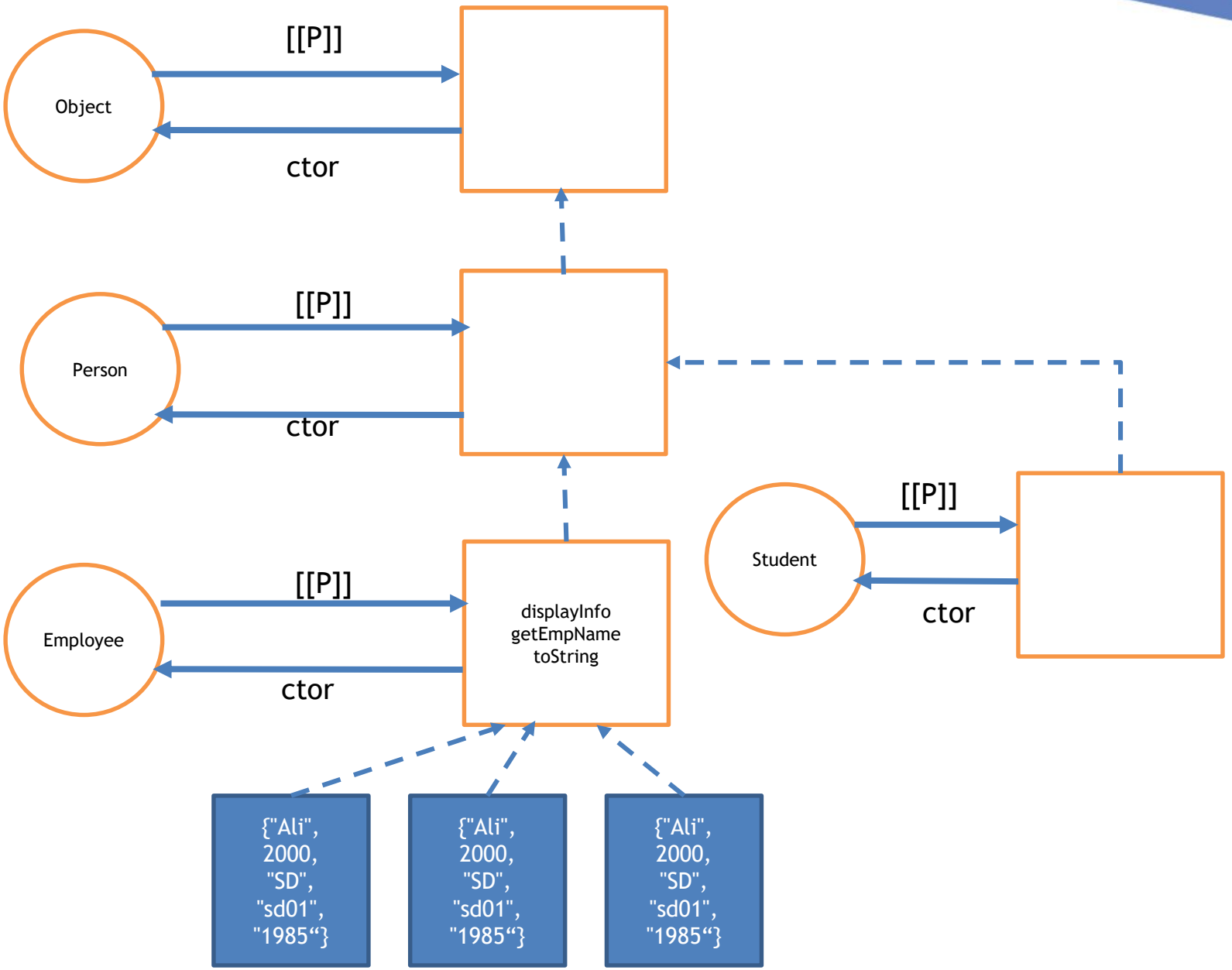
```
// prototype chaining
    Employee.prototype = Object.create(Person.prototype);

   var e = new Employee('Ahmed', 12);
   e.getName();    //Inherited from Person
```

Parasitic inheritance
is said to be
Pure Prototypal Inheritance.

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**.



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


Object
[[P]]
ctor
Person
[[P]]
ctor
Student
[[P]]
ctor
Employee
[[P]]
ctor
displayInfo
getEmpName
toString
{"Ali",
2000,
"SD",
"sd01",
"1985“}
{"Ali",
2000,
"SD",
"sd01",
"1985“}
{"Ali",
2000,
"SD",
"sd01",
"1985“}

Change your thinking from the class/inheritance design pattern to the behavior delegation design pattern.

Reminder

- Object & Object creation
  - Factory
  - Constructor
  - Literal Pattern
- this and new operator
- Class property and method
- Scope & Closure
  - Local scope variable = Private variable
  - Inner function = Private methods
- Privileged methods = getters & setters
- Property descriptor
  - Data descriptor
  - Accessor descriptor
- Prototype property
- Inheritance & delegation behavior

By the end of development phase

- Validating HTML
  - https://validator.w3.org/#validate_by_input
- Validating CSS
  - https://jigsaw.w3.org/css-validator/#validate_by_input
- Minifing & Compression
  - https://jscompress.com/
  - https://javascript-minifier.com/
  - https://www.minifier.org/
  - https://www.danstools.com/javascript-obfuscate/

Assignment

Assignment