Advanced JavaScript

Lecture 1

Object Oriented

- Encapsulation.
- Abstraction.
- Inheritance.
- Polymorphism.

Object Oriented

- **Encapsulation:**
  - We group related variables and methods into Object.
- **Abstraction:**
  - To hide Complexity and reduce the impact of changes.
- **Inheritance:**
  - To eliminate redundant code.
- **PolyMorphism (Many Forms):**
  - provides a way to perform a single action in different forms.

JavaScript objects

- ❑ If you understand objects, you understand JavaScript.
- ❑ In JavaScript, almost "everything" is an object.
- ❑ All JavaScript values, except primitives, are objects.
- ❑ Objects can store properties. Until now, a property was a simple “key-value” pair.
- ❑ JavaScript Objects are Mutable (They are addressed by reference, not by value).
- ❑ Example: JavaScript objects are mutable.

Objects - Creating Object

- ❑ Using an Object Literal
  - ❑ Example: Create object - Using an Object Literal

- ❑ Using the JavaScript Keyword new.
  - ❑ Example: Create object - Using the JavaScript Keyword new

Objects - Factory Function

- A regular function return an Object.
  - Example: Factory Function
  - Each time we call this factory, it will return a new instance of the created object.
  - Defining one factory in terms of another helps us break complex factories into smaller, reusable fragments. (in another words we can make nested factories).

Objects - Constructor function creation

- Constructor function technically is a regular function with some different conventions
    - Constructor function should start with capital letter.
    - Constructor function should executed only with new operator.
    - Example: Constructor functions - creation

Objects - Constructor function return

- Usually, we didn’t need to write return statement inside a constructor function , This will be returned automatically
- We can return object instead of This.
- Primitive will be ignored if you try to return it.
- Return with an Object returns that object , in all other cases this is returned.
- Example: Constructor functions - return

Objects - Why Constructor/ Factory function ?

- ❏ The regular { key: value } syntax is allow to create one object . but if you need to create many similar objects **Ex:** multiple users or menu items, the regular way is will be not suitable to this case.
- ❏ In another words: Use Constructor/Factory Function => If you need to implement reusable object creation code.

Objects - Add new property to object

- ❑ We can add new property using :
  - ❑ Dot notation => `Object.name = ‘Ahmed’`

  - ❑ Bracket notation => `Object[‘name’] = ‘Ahmed’;`

  - ❑ Defineproperty method => `Object.defineProperty(obj , ‘name’ , {value: ‘Ahmed’})`

Objects - defineProperty method.

- It defines a new property directly on an object, or modifies an existing property on an object, and returns the object.
- Syntax: `Object.defineProperty(obj, prop, descriptor)`
- By default, values added using `Object.defineProperty()` are immutable.
- Example: DefineProperty method

Objects - Descriptor

- The third parameter of `Object.defineProperty()` is an Object called Descriptor.
- Object descriptor have 4 attributes:
  - Value
  - Writable: if `true`, the value can be changed, otherwise it's read-only.
  - Enumerable: if `true`, then listed in loops, otherwise not listed
  - Configurable: if `true`, the property can be deleted and these attributes can be modified, otherwise not.
- The method `Object.getOwnPropertyDescriptor` allows to query the full information about a property.

Objects - Primitive and reference types

- Primitives types
  - Number
  - String
  - Boolean
  - Undefined
  - Null

- References are:
  - Object
  - Function
  - Array

- Primitives are copied by their value
- Objects are copied by their reference.

Objects - Private Properties

- ❑ If you want to make private variables (can not be access them from anywhere) => define them at function scope.
- ❑ In function scope (Local scope) the variables initialized when the function called and die once the execution finished.
- ❑ We can protect the object properties using setter / getter concept.
- ❑ Example: Setter / getter

Prototypal inheritance

- ❑ In javaScript, Objects have a special hidden property called [[prototype]] .
- ❑ That is either Null or reference to another Object .
- ❑ We used Prototypal inheritance to reuse the object without copy or reimplement its method.
- ❑ In another words: we use “Prototype” to take something and extend it.

Prototypal inheritance - How to use it ?

- There are many ways to implement prototypes in javaSCript
- __proto__ as a setter/getter.
- Note: __proto__ is not the same as [[prototype]] , is just a getter or setter for it.
- Example: Prototypal inheritance - using __Proto__

Prototypal inheritance - prototype

- ❑ The prototype chain can be longer Ex: Prototype chain.
- ❑ The references can't go in circles. JavaScript will throw an error if we try to assign `__proto__` in a circle.
- ❑ There can be only one `[[Prototype]]`. An object may not inherit from two others.
- ❑ The value of `This` is not affected by prototypes at all.

Prototypal inheritance - Loop

- ❑ If we need to iterate over object we use for..in .
- ❑ it iterates over inherited properties too.
- ❑ Object.keys(obj) : only returns own keys.
- ❑ If we'd like to exclude inherited properties, there's a built-in method obj.hasOwnProperty(key): it returns true if obj has its own (not inherited) property named key.

Prototype without __proto__

- The `__proto__` is considered outdated and somewhat deprecated.
- The modern methods are:
- `Object.create(proto[, descriptors])` – creates an empty object with given proto as `[[Prototype]]` and optional property descriptors. `Object.getPrototypeOf(obj)` – returns the `[[Prototype]]` of obj. `Object.setPrototypeOf(obj, proto)` – sets the `[[Prototype]]` of obj to proto.
- These should be used instead of `__proto__`.
- Example: Prototype without __proto__

Lab

- **Exercice 1:**
    - Create a constructor function Calculator that creates objects with 3 methods:
        - `read()` asks for two values using prompt and remembers them in object properties.
        - `sum()` returns the sum of these properties.
        - `mul()` returns the multiplication product of these properties.

Lab

❑ Exercice 2:

- ❑ Create a Stopwatch object using constructor function.
- ❑ The stopwatch object have 1 property called duration (Intility duration is 0) and 3 methods: start() , stop() and reset().
- ❑ If you call start() method for the first time => the watch should be start to count .
- ❑ Note: you can't call start() twice :
  If you call start() again (throw an error the watch is already started).

Lab

- ❏ **Exercice 2:**
  - ❏ If you call stop() method => the watch should be stope to count .
  - ❏ Note: you can't call stop() twice :
    If you call stop() again (throw an error the watch is not started yet).
  - ❏ Note: if you call start() again after you has been stop it the duration must continue count from last value.
  - ❏ If you call reset() method => this should reset the duration to the initial state.

Lab

- **Exercice 3:**
  - Create an object called Teacher derived from the Person object
  - implement a method called teach which receives a string called subject, and prints out:
    [teacher's name] is now teaching [subject]
  - Note: Person Object has 2 properties: name and age received dynamically .