TypeScript Class Members

1. Constructor
2. Properties
3. Methods
4. Accessors

Constructor

- Constructor is a special type of method in a class that executes automatically when a class is loaded into memory.
- Generally, a method executes when it is called explicitly.
- Constructor doesn't require an explicit call.
- Constructor is required to handle implicit actions for any class.
- TypeScript constructor is defined by using "constructor" keyword.
- Constructor is an anonymous method. It is a method without any name defined.
- Constructor implicitly uses the class name.
- Every class by default have a constructor. You can configure an explicit constructor.

```
Class Demo
{

}

let obj = new Demo;   → Calls constructor
let obj = new Demo(); → Explicit constructor call to pass parameters
```

- Constructor will be called implicitly for every object only once.
- Constructor is called at the time of loading class into memory.

Ex:

```
class Database

{

   constructor(){
```

```
        console.log("Connected to Database");
    }
    public Insert() {
        console.log("Record Inserted");
    }
}
let oracle = new Database;
oracle.Insert();
oracle.Insert();
let mysql = new Database;
mysql.Insert();
```

```
Connected to Database
Record Inserted
Record Inserted
Connected to Database
Record Inserted
```

- Constructor can be parameter less or parameterized.
- Parameterized constructor is used to append the implicit functionality.
- A parameter less constructor will perform the same functionality across any number of objects and at any state.
- A parameterized constructor can modify the functionality.
  Syntax:
  Class Demo
  {
  constructor(param:Type) { }
  }
- Every parameter defined in constructor is mandatory.
- You have to pass arguments into constructor at the time of allocating memory for class.

Syntax:

let obj = new Demo(); // Invalid

let obj = new Demo; // Invalid

let obj = new Demo(args); // valid

- You can configure the constructor parameters as optional by using null reference character “?”.

  Syntax:

  constructor(param?:Type) { }

Ex:

```
class Database
{
   constructor(dbName?:string){
      if(dbName==undefined) {
         console.log(`Connected to UnKnown Database`);
      } else {
      console.log(`Connected to ${dbName} Database`);
      }
   }
   public Insert() {
      console.log("Record Inserted");
   }
}
let oracle = new Database("Oracle");
oracle.Insert();
let mysql = new Database;
mysql.Insert();
```

```
Connected to Oracle Database
Record Inserted
Connected to UnKnown Database
Record Inserted
```

- A constructor can have multiple parameters.
- *But a required parameter can't follow optional parameter.*
  Syntax:
  constructor(dbName?: string, server: string) { } // invalid

Ex:

```
class Database
{
   constructor(server:string, dbName?:string){
      console.log(`Connect to ${server} Server`);
      if(dbName==undefined) {
         console.log(`Connected to UnKnown Database`);
      } else {
      console.log(`Connected to ${dbName} Database`);
      }
   }
   public Insert() {
      console.log("Record Inserted");
   }
}
let oracle = new Database("Oracle","EmployeesDb");
oracle.Insert();
let mysql = new Database("MySql");
```

mysql.Insert();

- Constructor can be defined with array type parameter to pass and multiple values.
- To pass values into constructor you have to configure an array type memory dynamically or you can create an array and pass array into constructor.

  Ex:

```
class Database
{
   constructor(commands:string[]){
      for(var item of commands){
         console.log(item);
      }
   }
}
let oracle = new Database(["Insert","Update","Delete"]);
                                (or)
let oracle = new Database(new
Array("Insert","Update","Delete"));
```

- A constructor can be defined with multiple Array parameters.

  Ex:

```
class Database
{
   constructor(commands:string[], tables:string[]){
      for(var item of commands){
         console.log(item);
      }
      for(var item of tables){
         console.log(item);
      }
   }
```

```
}
let oracle = new Database(new
Array("Insert","Update","Delete"),new
Array("Employee","Products"));
```

- You can configure Array parameters along with other parameters.

```
Ex:
class Database
{
   constructor(commands:string[], count:number){
      for(var item of commands){
         console.log(item);
      }
      console.log(`Total Count= ${count}`)
   }
}
let oracle = new Database(new
Array("Insert","Update","Delete"),12);
```

- There is no order dependency in configuring Array parameters.
- **ES5 introduced Rest Parameters, which TypeScript can use for constructor or methods.**
- Rest parameter is a single parameter which can store multiple arguments.
- A Rest parameter allows to pass multiple values into constructor or method so that they can stored in a single reference.
- Rest parameter can be defined by using "…"
- Every constructor or method can have only **one rest parameter**.
- Rest parameter must be the last parameter in formal parameters.

Ex:

```
class Database
{
   constructor(count:number, ...commands:string[]){
      for(var item of commands){
         console.log(item);
      }
      console.log(`Total Count= ${count}`)
   }
}
let obj = new Database(12,"Insert","Update","Delete");
```

- A constructor can be defined with Object Type parameter

Ex:

```
class Database
{
   constructor(product:any){
     for(var property in product) {
        console.log(`${property}:${product[property]}`)
     }
   }
}
let obj = new Database({Id:1, Name:"Mobile"});
```

- Constructor can allow to pass a function as parameter.

Ex:

```
class Database
{
   constructor(pwd, success:any, failure:any){
      if(pwd=="tiger") {
         console.log(success());
```

```
      } else {
         console.log(failure());
      }
   }
}
let oracle = new Database("tigers", function(){return
"Connection Successfull"}, function(){return "Invalid
Password"});
```

- **A constructor must have a super call in side derived class.**
- If a class is extended then the derived class constructor must have a super call.
- Derived class constructor can't execute before the super class constructor. Hence a derived class constructor must have super call. "super()"

Ex:

```
class SuperClass
{
   constructor() {
      console.log(`Super Class Constructor`);
   }
}
class Derived extends SuperClass
{
   constructor(){
      super();
      console.log(`Derived Class Constructor`);
   }
}
let obj = new Derived();
```

- **A constructor can be defined with access modifiers**

- Always a constructor must be public in access in order to allow instantiation (Creating of Object)
- Private constructor will not allow to extend the class. And will not allow to create an instance.
- Products constructor will allow extensibility but will not allow to create an instance.
- Constructor in TypeScript can’t be static.

Properties

- Property is a named member of class.
- Properties are used to store data.
- The memory is allocated when the class is loaded into memory.
- Property can have restricted access.
- Property is accessible by using accessor which allow to set and get value dynamically.
- Class can store data only in properties.

Syntax:

```
class className
{
  accessModifier propertyName = value;
}
```

TypeScript support Type Inference for properties. The data type will be determined according to the value assigned.

- You can initialize and render values into a property.

Ex:

```
class Product
{
   public Name:string = "TV";              // Initialization
```

```
    public Price:number;
    public InStock:boolean;
}
let tv = new Product();
tv.Name = "Samsung TV";                    // Rendering
tv.Price = 45000.55;
tv.InStock = true;
```

- You can restrict rendering values into a property by using “read-only”

Ex:

```
class Product
{
    public readonly Name:string = "TV";
    public Price:number;
    public InStock:boolean;
}
let tv = new Product();
tv.Name = "Samsung TV";  // Invalid
tv.Price = 45000.55;
tv.InStock = true;
```

- You can configure authorized access to any property by using “Accessors”

Accessors

- TypeScript supports getters and setters as a way of intercepting access to a member of an object.
- Accessors will give fine grained control over how a member is accessed.
- It can provide authorized access to properties.

- It allows to set value for a situation and the same can restrict value for another situation.
- TypeScript accessors are defined by using
    - get()
    - set()

Syntax:

```
get accessorName() {
    return propertyValue;
}
set accessorName(val){
    this.property = val;
}
```

Ex:

1. Add a new TypeScript file by name “demo.ts”

```
let userName:string = prompt("User Name");
let password:string = prompt("Password");

class Product
{
  private _productName:string;

  get ProductName():string {
    return this._productName;
  }

  set ProductName(newName:string){
    if(userName=="john" && password=="admin")
    {
```

```
            this._productName = newName;
        }
        else
        {
            alert("Error : You are not authorized to Set Product
Name");
        }
    }
}
let tv = new Product();
tv.ProductName = "Samsung TV";
if(tv.ProductName) {
    document.write(`Name=${tv.ProductName}`);
}
```

2. Compile “demo.ts”
   > tsc demo.ts
3. Create a new HTML page “home.html” and link the “demo.js”
   ```
   <script src="demo.js">
   </script>
   ```
4. Run with live server.

Functions and Methods