Constructor in ES6 Classes

- JavaScript uses special functions called constructor functions to define and initialize objects and their features.
- Classes have a special method called constructor which gets invoked when a class is initialized via new keyword.
- In simple words, a constructor is a function which automatically runs anytime when an object is instantiated from that class.
- Constructor is just a function which gets called automatically when we create an instance of the class.
- Constructors are nothing new. Calling any function with the "new" keyword causes it to return an object this is called making a constructor call, and such functions are generally called constructors.
- Constructors are responsible for allocating memory for the objects of the class.
- There can only be one special method with the name "constructor" in a class. A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.
- A constructor enables us to provide any custom initialization that must be done before any other methods can be called on an instantiated object.
- If we don't provide our own constructor, then a default constructor will be supplied for our class. If our class is a base class, the default constructor is empty, e.g., "constructor() {}"
- If our class is a derived class, the default constructor calls the parent constructor, passing along any arguments that were provided(the case of inheritance).

Significance of constructor function

- Constructor functions essentially eliminate the need to create separate object literals for similar tasks.
- Constructors are useful because we'll often come across situations in which we don't know how many objects we will be creating.
- Constructors provide the means to create as many objects as we need in an effective way.
- Constructor is commonly used to set up some default data or initialize needed variables.
- Responsible for allocating memory for the objects of the class using new.

What happens when a constructor is called?

In JavaScript, here's what happens when a constructor is invoked:

- A new empty object is created using new.
- "this" keyword starts referring to that newly created object and hence it becomes the current instance object.
- The newly created object is then returned as the constructor's returned value.

Points to be remember for constructor function

- The purpose of a constructor is to create an object and set values if there are any object properties present.
- We never actually call the constructor itself since it is automatic.
- There is no comma after the closing curly brace of the constructor.
- Constructors may be invoked either as a "function call" or by a "new" operator.
- JavaScript doesn't support constructor overloading. Therefore you can't have more than one constructor, within your class, or else the JavaScript compiler will throw an error.
- We cannot make a class constructor static in JavaScript via the static keyword, but we can make class members as static using the keyword.
- The constructor doesn't return a value is because it's not called directly by your code, it's called by the memory allocation and object initialization code in the runtime. Its return value (if it actually has one when compiled down to machine code) is opaque to the user - therefore, you can't specify it.

Types of constructors

There are two types of constructors :-

- Built-in constructors such as Array and Object, which are available automatically in the execution environment at runtime.
- Custom constructors, which define properties and methods for our own type of object.

Syntax of constructor method

Syntax 1 - Default Constructor (Zero Parameter) in JavaScript

```
class className{
   constructor(){
      // Instance Variables / Properties
   }
}
```

Syntax 2 - Parameterized Constructor in JavaScript

```
class className{
   constructor(param1,param2,....paramN){
      // Instance Variables / Properties
   }
}
```

Types of constructors in ES6 classes

There are two types of constructors in es6 classes :-

a) Default Constructor (Zero Parameter)

- A constructor without any parameters is known as default constructor which gets created when we don't define any constructor.
- If we don't provide our own constructor, then a default constructor will be supplied for our class.
- In Default constructor, the instance object is created using "new" keyword and class has been called like function.
- Whenever we calls / access an instance object with no parameters then a default constructor will automatically call.

Example -- Automatically Runs Constructor

```
class Vehicle {
    constructor() {
        console.log('Running Automatically');
    }
}
 let vehicle = new Vehicle();
```

Example -- Checking Constructor Content

```
class Meetup {
    constructor() {
        console.log("It's not a Default Constructor")
    }
}
console.log(Meetup.toString());
```

Example -- Class Without Constructor

```
class Person {

   /* JavaScript inserts something like this:
    constructor () { }          // which would be default constructor
   */

}
let info = new Person();
console.log(info); // Person {}
```

Example -- Using "this" keyword

```
class Person {
   constructor() {
      console.log(this); // Person{}
   }
}
let p = new Person();
console.log(p)
```

Example -- Overwriting Default Constructor using our own constructor*

```
class Person {
   constructor() {
      console.log("Hello I'm overwriting Default Constructor");
   }
}
let info = new Person();
console.log(info);
```

b) Parameterized Constructor

- A parameterized constructor enables us to provide any custom initialization that must be done before any other methods can be called on an instantiated object.
- In parameterized constructor the instance object is created using "new" keyword and class has been called like function.
- After that, we can pass arguments to the class as well like any other function and behind the scene, it will call the constructor function for initializing the instance variables name and location.

Example -- Calling Constructor Directly

```
class Meetup {
   constructor(name, location) {
      this.name = name;
      this.location = location;
   }
}
console.log(new Meetup.prototype.constructor());
```

Example -- Calling Constructor Directly

```
class Person{
   constructor(name,age){
      this.fullName = name; // Instance Property
      this.newAge = age; // Instance Property
   }

}
console.log(new Person.prototype.constructor("Rohit",27))
```

Example -- Constructor Content

```
class Meetup {
   constructor(name, location) {
      this.name = name;
      this.location = location;
   }
}
console.log(Meetup.toString())
```

Example -- Constructor with single parameter

```
class Display {
   constructor(name) {
      this.name = name
   }
}
let display = new Display("Rohit");
console.log(display);
```

Example -- Constructor with multiple parameter

```
class Meetup {
   constructor(name, location) {
      this.name = name;
      this.location = location;
   }
}
let jsMeetup = new Meetup('JS', 'Banglore');
let ngMeetup = new Meetup('Angular', 'Noida');
console.log(jsMeetup);
console.log(ngMeetup);
```

'constructor' property

- The constructor property returns a reference to the Object constructor function that created the instance object.
- The value of the 'constructor' property is a reference to the function itself, not a string containing the function's name.
- Every object in JavaScript inherits a "constructor" property from its prototype, which points to the constructor function that has created the object.

Example

```
class Person {
   constructor() {
      console.log(this)
   }
}
let p = new Person();
console.log(p.constructor == Person);
```

Example

```
var s = new String("text");
s.constructor === String
"text".constructor === String
```

Example

```
var o = new Object();
o.constructor === Object
var o = {};
o.constructor === Object
```

Example

```
var a = new Array();
a.constructor === Array
[].constructor === Array
```

"new.target" property

- The "new.target" pseudo-property detects whether a function or constructor gets called using the "new" operator.
- The "new.target" pseudo-property is available in all functions.
- In class constructors, it refers to the constructed class.
- When we call an ordinary function with the "new" keyword, the value of "new.target" within the function body is the function itself but If the function wasn't called with "new" keyword, its value is undefined.
- In arrow functions, "new.target" is inherited from the surrounding scope.

Example -- In Functions

```
function Foo() {
  if (!new.target) {
      throw 'Foo() must be called with new'
      }
      console.log('Foo instantiated with new')
}
new Foo()
Foo()
```

Example -- In Classes

```
class Person {
   constructor() {
      console.log(new.target.name);
   }
}
let p = new Person();
console.log(p)
}
```

Instance members in constructor

- The properties & methods we define inside of a class constructor are knowns as instance members.
- The properties we define inside of a class constructor are known as instance properties.

Instance Variables

- Instance variables get created and initialized using constructor.
- Instance variables are nothing but called properties of the object.
- We can declare instance variables inside of class constructor by using "this" keyword.
- The instance variables can only be accessed by the object instance.

Example

```
class Person {
    constructor() {
        this.name = "Rohit"; // Here, this.name is the instance property
        this.age = 27    // Here, this.age is the instance property
    }
}
let p = new Person();
console.log(p.name); // Rohit
console.log(p.age); // 27
```

Example

```
class Employee {
    constructor() {
        this.Name = "Rohit Singh";
    }
}
const emp = new Employee();
console.log(emp.Name);
console.log(Employee.Name);
```

Instance Properties / Instance Fields

a) Instance Properties

- Instance property must be defined inside of class methods and can be accessed via 'this' keyword inside of class.
- The 'this' keyword refers to the object created by the class (the instance).
- All instance property is accessible only via class instance objects.

Example

```
class Employee {
   constructor() {
     this.Name = "Rohit Singh";
   }
}
const emp = new Employee();
console.log(emp.Name);
console.log(Employee.Name);
```

Example

```
class Employee {
   constructor(name,age) {
     this.getName = name;
     this.getAge = age;
   }
}
let emp = new Employee("Rohit",27);
console.log(emp.getName)
console.log(emp.getAge)
```

Public Properties / Public Fields

- These members of the class and available to everyone that can access the (owner) class instance.
- Public data members must be defined outside of class methods.
- Public data members always available for every created instance of a class.

Example

```
class Person{
   fullName = "Rohit Singh"; // Public Data Members
}
let p = new Person();
console.log(p.fullName)
```

Example

```
class Person{
   fullName = "Rohit Singh"; // Public Data Members
}
let p = new Person();
console.log(p.fullName)
p.fullName = "Rahul Singh";
console.log(p.fullName)
```

Example

```
class Person{
   name = "Yash"; // Public Data Members
   age = "Singh"; // Public Data Members
   constructor(name,age){
      this.fullName = name; // Instance Data Members
      this.newAge = age; // Instance Data Members
   }
}
let p = new Person("Rohit","Singh")
console.log(p.name) // Yash
console.log(p.fullName) // Rohit
```

Prototype Properties / Prototype Fields

- A prototype is a special kind of object by which additional properties can attach to the 'class' and it will share across all the instances of that class.
- We can create 'prototype' properties outside of the class.
- The 'prototype' properties can be accessed via 'Instance Objects' as well as with 'Classes'.

Example

```
class Person{
   name = "Yash"; // Public Property
   age = "Singh"; // Public Property
   constructor(name,age){
      this.fullName = name; // Instance Property
      this.newAge = age; // Instance Property
   }
}
Person.prototype.gender = "Male" // Prototype Property
let p = new Person("Rohit","Singh")
console.log(p.gender)
console.log(Person.prototype.gender)
```

Private Properties / Private Fields

- Private properties are only accessible within the class that instantiated the object.
- We can create private properties inside of class by using '#' followed by property name.
- They are accessible only inside the class.

Example

```
class Person {
   #fullName = "Rohit Singh";
   prntName(){
      return this.#fullName
   }
}
let p = new Person();
console.log(p.prntName())
```

Example

```
class Person{
   #getName;
   #getAge;
   constructor(name,age) {
      this.#getName = name;
      this.#getAge = age;
   }
   prntName(){
      return `Hello ${this.#getName} your age is ${this.#getAge}`;
   }
}
const p = new Person("Rohit Singh",27);
console.log(p.prntName());
```

Protected Data Members

- A protected member is accessible within the class (similar to private) and any object that inherits from it.
- Protected members of a class are accessible from within the class and are also available to its sub-classes.
- The Protected properties are usually prefixed with an underscore '_'.
- Those data member that are read only are known as protected data members.
- Protected fields are not implemented in JavaScript on the language level, but in practice they are very convenient, so they are emulated.
- Protected fields are naturally inheritable.

Static Properties / Class Properties / Static Fileds

- Static properties are also possible, they look like regular class properties, but prepended by 'static' keyword.
- The "static" keyword defines a static property for a class.
- Static property can be accessed without instantiating their class and cannot be called through a class instance.
- Static properties cannot be accessed by the instances of the class but they can be accessed by the class itself.
- Static properties are useful for caches, fixed-configuration, or any other data you don't need to be replicated across instances.

Example

```
class Person{
   static myName = "Rohit Singh";
}
let p = new Person()
console.log(p.myName)
console.log(Person.myName)
```

Example

```
class Person {
   static myName = "Rohit Singh";
}
console.log(Person.myName);
Person.myName = "Rahul Singh";
console.log(Person.myName)
```