ES6 - Classes

Object Orientation is a software development paradigm that follows real-world modelling. Object Orientation, considers a program as a collection of objects that communicates with each other via mechanism called **methods**. ES6 supports these object-oriented components too.

Object-Oriented Programming Concepts

To begin with, let us understand

- **Object** − An object is a real-time representation of any entity. According to Grady Brooch, every object is said to have 3 features −
  - **State** − Described by the attributes of an object.
  - **Behavior** − Describes how the object will act.
  - **Identity** − A unique value that distinguishes an object from a set of similar such objects.
- **Class** − A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
- **Method** − Methods facilitate communication between objects.

Let us translate these Object-Oriented concepts to the ones in the real world. For example: A car is an object that has data (make, model, number of doors, Vehicle Number, etc.) and functionality (accelerate, shift, open doors, turn on headlights, etc.)

Prior to ES6, creating a class was a fussy affair. Classes can be created using the class keyword in ES6.

Classes can be included in the code either by declaring them or by using class expressions.

Syntax: Declaring a Class

```
class Class_name {
}
```

Syntax: Class Expressions

```
var var_name = new Class_name {
}
```

The class keyword is followed by the class name. The rules for identifiers (already discussed) must be considered while naming a class.

A class definition can include the following −

- **Constructors** − Responsible for allocating memory for the objects of the class.
- **Functions** − Functions represent actions an object can take. They are also at times referred to as methods.

These components put together are termed as the data members of the class.

Note − A class body can only contain methods, but not data properties.

Example: Declaring a class

```
class Polygon {
   constructor(height, width) {
      this.height = height;
      this.width = width;
   }
}
```

Example: Class Expression

```
var Polygon = class {
   constructor(height, width) {
      this.height = height;
      this.width = width;
   }
}
```

The above code snippet represents an unnamed class expression. A named class expression can be written as.

```
var Polygon = class Polygon {
   constructor(height, width) {
      this.height = height;
```

```
      this.width = width;
   }
}
```

Note − Unlike variables and functions, classes cannot be hoisted.

Creating Objects

To create an instance of the class, use the new keyword followed by the class name. Following is the syntax for the same.

```
var object_name= new class_name([ arguments ])
```

Where,

- The new keyword is responsible for instantiation.
- The right hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.

Example: Instantiating a class

```
var obj = new Polygon(10,12)
```

Accessing Functions

A class's attributes and functions can be accessed through the object. Use the '.' **dot notation** (called as the period) to access the data members of a class.

```
//accessing a function
obj.function_name()
```

Example: Putting them together

```
'use strict'
class Polygon {
   constructor(height, width) {
```

```
      this.h = height;
      this.w = width;
   }
   test() {
      console.log("The height of the polygon: ", this.h)
      console.log("The width of the polygon: ",this. w)
   }
}

//creating an instance
var polyObj = new Polygon(10,20);
polyObj.test();
```

The Example given above declares a class 'Polygon'. The class's constructor takes two arguments - height and width respectively. The **'this'** keyword refers to the current instance of the class. In other words, the constructor above initializes two variables h and w with the parameter values passed to the constructor. The **test ()** function in the class, prints the values of the height and width.

To make the script functional, an object of the class Polygon is created. The object is referred to by the **polyObj** variable. The function is then called via this object.

The following output is displayed on successful execution of the above code.

```
The height of the polygon:  10
The width of the polygon:  20
```

Setters and Getters

Setters

A setter function is invoked when there is an attempt to set the value of a property. The **set keyword** is used to define a setter function. The syntax for defining a setter function is given below −

```
{set prop(val) { . . . }}
{set [expression](val) { . . . }}
```

prop is the name of the property to bind to the given function. **val** is an alias for the variable that holds the value attempted to be assigned to property. **expression** with ES6, can be used as a property name to bind to the given function.

Example

```
<script>
   class Student {
      constructor(rno,fname,lname){
         this.rno = rno
         this.fname = fname
         this.lname = lname
         console.log('inside constructor')
      }
      set rollno(newRollno){
         console.log("inside setter")
         this.rno = newRollno
      }
   }
   let s1 = new Student(101,'Sachin','Tendulkar')
   console.log(s1)
   //setter is called
   s1.rollno = 201
   console.log(s1)
</script>
```

The above example defines a class Student with **three properties** namely **rno, fname and lname**. A setter function **rollno()** is used to set the value for the rno property.

The output of the above code will be as shown below −

```
inside constructor
Student {rno: 101, fname: "Sachin", lname: "Tendulkar"}
inside setter
Student {rno: 201, fname: "Sachin", lname: "Tendulkar"}
```

Example

The following example shows how to use an **expression** as a property name with a **setter function**.

```
<script>
   let expr = 'name';
      let obj = {
      fname: 'Sachin',
      set [expr](v) { this.fname = v; }
   };
   console.log(obj.fname);
   obj.name = 'John';
   console.log(obj.fname);
</script>
```

The output of the above code will be as mentioned below −

```
Sachin
John
```

Getters

A **getter function** is invoked when there is an attempt to fetch the value of a property. The **get keyword** is used to define a getter function. The syntax for defining a getter function is given below −

```
{get prop() { ... } }
{get [expression]() { ... } }
```

prop is the name of the property to bind to the given function.

expression − Starting with ES6, you can also use expressions as a property name to bind to the given function.

Example

```
<script>
   class Student {
      constructor(rno,fname,lname){
         this.rno = rno
```

```
         this.fname = fname
         this.lname = lname
         console.log('inside constructor')
      }
      get fullName(){
         console.log('inside getter')
         return this.fname + " - "+this.lname
      }
   }
   let s1 = new Student(101,'Sachin','Tendulkar')
   console.log(s1)
   //getter is called
   console.log(s1.fullName)
</script>
```

The above example defines a class Student with three properties namely **rno, fname and lname**. The getter function **fullName()** concatenates the **fname** and **lname** and returns a new string.

The output of the above code will be as given below −

```
inside constructor
Student {rno: 101, fname: "Sachin", lname: "Tendulkar"}
inside getter
Sachin - Tendulkar
```

Example

The following example shows how to use an expression as a property name with a getter function −

```
<script>
   let expr = 'name';
   let obj = {
      get [expr]() { return 'Sachin'; }
   };
   console.log(obj.name);
</script>
```

The output of the above code will be as mentioned below −

```
Sachin
```

The Static Keyword

The static keyword can be applied to functions in a class. Static members are referenced by the class name.

Example

```
'use strict'
class StaticMem {
   static disp() {
      console.log("Static Function called")
   }
}
StaticMem.disp() //invoke the static metho
```

Note − It is not mandatory to include a constructor definition. Every class by default has a constructor by default.

The following output is displayed on successful execution of the above code.

```
Static Function called
```

The instanceof operator

The instanceof operator returns true if the object belongs to the specified type.

Example

```
'use strict'
class Person{ }
var obj = new Person()
var isPerson = obj instanceof Person;
console.log(" obj is an instance of Person " + isPerson);
```

The following output is displayed on successful execution of the above code.

```
obj is an instance of Person True
```

Class Inheritance

ES6 supports the concept of **Inheritance**. Inheritance is the ability of a program to create new entities from an existing entity - here a class. The class that is extended to create newer classes is called the **parent class/super class**. The newly created classes are called the **child/sub classes**.

A class inherits from another class using the 'extends' keyword. Child classes inherit all properties and methods except constructors from the parent class.

Following is the syntax for the same.

```
class child_class_name extends parent_class_name
```

Example: Class Inheritance

```
'use strict'
class Shape {
   constructor(a) {
      this.Area = a
   }
}
class Circle extends Shape {
   disp() {
      console.log("Area of the circle:  "+this.Area)
   }
}
var obj = new Circle(223);
obj.disp()
```

The above example declares a class Shape. The class is extended by the Circle class. Since, there is an inheritance relationship between the classes, the child class i.e., the class Circle gets an implicit access to its parent class attribute i.e., area.

The following output is displayed on successful execution of the above code.

```
Area of Circle: 223
```

Inheritance can be classified as −

- **Single** − Every class can at the most extend from one parent class.
- **Multiple** − A class can inherit from multiple classes. ES6 doesn't support multiple inheritance.
- **Multi-level** − Consider the following example.

```
'use strict'
class Root {
   test() {
      console.log("call from parent class")
   }
}
class Child extends Root {}
class Leaf extends Child

//indirectly inherits from Root by virtue of inheritance {}
var obj = new Leaf();
obj.test()
```

The class Leaf derives the attributes from the Root and the Child classes by virtue of multilevel inheritance.

The following output is displayed on successful execution of the above code.

```
call from parent class
```

Class Inheritance and Method Overriding

Method Overriding is a mechanism by which the child class redefines the superclass method. The following example illustrates the same −

```
'use strict' ;
class PrinterClass {
   doPrint() {
```

```
      console.log("doPrint() from Parent called… ");
   }
}
class StringPrinter extends PrinterClass {
   doPrint() {
      console.log("doPrint() is printing a string…");
   }
}
var obj = new StringPrinter();
obj.doPrint();
```

In the above Example, the child class has changed the superclass function's implementation.

The following output is displayed on successful execution of the above code.

```
doPrint() is printing a string…
```

The Super Keyword

ES6 enables a child class to invoke its parent class data member. This is achieved by using the **super** keyword. The super keyword is used to refer to the immediate parent of a class.

Consider the following example −

```
'use strict'
class PrinterClass {
   doPrint() {
      console.log("doPrint() from Parent called…")
   }
}
class StringPrinter extends PrinterClass {
   doPrint() {
      super.doPrint()
      console.log("doPrint() is printing a string…")
   }
}
var obj = new StringPrinter()
obj.doPrint()
```

The **doPrint()** redefinition in the class StringWriter, issues a call to its parent class version. In other words, the super keyword is used to invoke the doPrint() function definition in the parent class - PrinterClass.

The following output is displayed on successful execution of the above code.

```
doPrint() from Parent called.
doPrint() is printing a string.
```