ES6 Classes

Classes are an essential part of object-oriented programming (OOP). Classes are used to define the blueprint for real-world object modeling and organize the code into reusable and logical parts.

Before ES6, it was hard to create a class in [JavaScript](https://www.javatpoint.com/javascript-tutorial). But in ES6, we can create the class by using the **class** keyword. We can include classes in our code either by class expression or by using a class declaration.

A class definition can only include **constructors** and **functions**. These components are together called as the data members of a class. The classes contain **constructors** that allocates the memory to the objects of a class. Classes contain **functions** that are responsible for performing the actions to the objects.

**Syntax: Class Expression**

1. var var\_name = **new** class\_name {
2. }

**Syntax: Class Declaration**

1. **class** Class\_name{
2. }

Let us see the illustration for the class expression and class declaration.

**Example - Class Declaration**

1. **class** Student{
2. constructor(name, age){
3. **this**.name = name;
4. **this**.age = age;
5. }
7. }

**Example - Class Expression**

1. var Student = **class**{
2. constructor(name, age){
3. **this**.name = name;
4. **this**.age = age;
5. }
6. }

Instantiating an Object from class

Like other object-oriented programming languages, we can instantiate an object from class by using the new keyword.

**Syntax**

1. var obj\_name = **new** class\_name([arguements])

**Example**

1. var stu = **new** Student('Peter', 22)

Accessing functions

The object can access the attributes and functions of a class. We use the **'.' dot notation (**or **period)** for accessing the data members of the class.

**Syntax**

1. obj.function\_name();

**Example**

1. 'use strict'
2. **class** Student {
3. constructor(name, age) {
4. **this**.n = name;
5. **this**.a = age;
6. }
7. stu() {
8. console.log("The Name of the student is: ", **this**.n)
9. console.log("The Age of the student is: ",**this**. a)
10. }
11. }
13. var stuObj = **new** Student('Peter',20);
14. stuObj.stu();

In the above example, we have declared a class **Student**. The constructor of the class contains two arguments **name** and **age,** respectively. The keyword **'this'** refers to the current instance of the class. We can also say that the above constructor initializes two variables **'n' and 'a'** along with the parameter values passed to the constructor.

The function **stu()** in the class will print the values of **name** and **age**.

**Output**

The Name of the student is: Peter

The Age of the student is: 20

Note: Including a constructor definition is mandatory in class because, by default, every class has a constructor.

The Static keyword

The **static** keyword is used for making the static functions in the class. Static functions are referenced only by using the class name.

**Example**

1. 'use strict'
2. **class** Example {
3. **static** show() {
4. console.log("Static Function")
5. }
6. }
7. Example.show() //invoke the static method

**Output**

Static Function

Class inheritance

Before the [ES6](https://www.javatpoint.com/es6), the implementation of inheritance required several steps. But ES6 simplified the implementation of inheritance by using the **extends** and **super** keyword.

Inheritance is the ability to create new entities from an existing one. The class that is extended for creating newer classes is referred to as **superclass/parent** class, while the newly created classes are called **subclass/child class**.

A class can be inherited from another class by using the **'extends'** keyword. Except for the constructors from the parent class, child class inherits all properties and methods.

**Syntax**

1. **class** child\_class\_name extends parent\_class\_name{
2. }

A class inherits from the other class by using the **extends** keyword.

**Example**

1. 'use strict'
2. **class** Student {
3. constructor(a) {
4. **this**.name = a;
5. }
6. }
7. **class** User extends Student {
8. show() {
9. console.log("The name of the student is:  "+**this**.name)
10. }
11. }
12. var obj = **new** User('Sahil');
13. obj.show()

In the above example, we have declared a class **student**. By using the **extends** keyword, we can create a new class **User** that shares the same characteristics as its parent class **Student**. So, we can see that there is an inheritance relationship between these classes.

**Output**

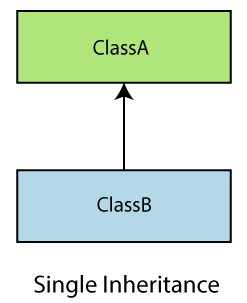
The name of the student is: Sahil

Types of inheritance

Inheritance can be categorized as Single-level inheritance, Multiple inheritance, and Multi-level inheritance. Multiple inheritance is not supported in ES6.

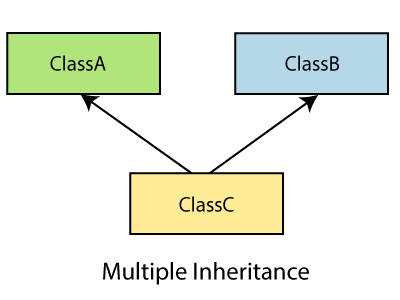
Single-level Inheritance

It is defined as the inheritance in which a derived class can only be inherited from only one base class. It allows a derived class to inherit the behavior and properties of a base class, which enables the reusability of code as well as adding the new features to the existing code. It makes the code less repetitive.



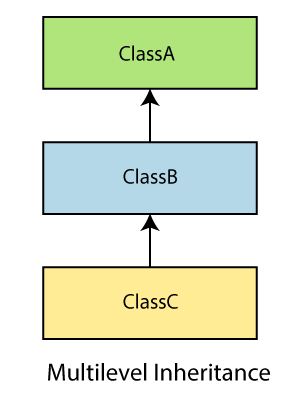
Multiple Inheritance

In multiple inheritance, a class can be inherited from several classes. It is not supported in ES6.



Multi-level Inheritance

In Multi-level inheritance, a derived class is created from another derived class. Thus, a multi-level inheritance has more than one parent class.



Let us understand it with the following example.

**Example**

1. **class** Animal{
2. eat(){
3. console.log("eating...");
4. }
5. }
6. **class** Dog extends Animal{
7. bark(){
8. console.log("barking...");
9. }
10. }
11. **class** BabyDog extends Dog{
12. weep(){
13. console.log("weeping...");
14. }
15. }
16. var d=**new** BabyDog();
17. d.eat();
18. d.bark();
19. d.weep();

**Output**

eating...

barking...

weeping...

Method Overriding and Inheritance

It is a feature that allows a child class to provide a specific implementation of a method which has been already provided by one of its parent class.

There are some rules defined for method overriding -

* The method name must be the same as in the parent class.
* Method signatures must be the same as in the parent class.

Let us try to understand the illustration for the same:

**Example**

1. 'use strict' ;
2. **class** Parent {
3. show() {
4. console.log("It is the show() method from the parent class");
5. }
6. }
7. **class** Child extends Parent {
8. show() {
9. console.log("It is the show() method from the child class");
10. }
11. }
12. var obj = **new** Child();
13. obj.show();

In the above example, the implementation of the superclass function has changed in the child class. You will get the following output after the successful execution of the above code:

**Output**

It is the show() method from the child class

The super keyword

It allows the child class to invoke the properties, methods, and constructors of the immediate parent class. It is introduced in ECMAScript 2015 or ES6. The **super.prop** and **super[expr]** expressions are readable in the definition of any method in both object literals and classes.

**Syntax**

super(arguments);

**Example**

In this example, the characteristics of the parent class have been extended to its child class. Both classes have their unique properties. Here, we are using the **super** keyword to access the property from parent class to the child class.

1. 'use strict' ;
2. **class** Parent {
3. show() {
4. console.log("It is the show() method from the parent class");
5. }
6. }
7. **class** Child extends Parent {
8. show() {
9. super.show();
10. console.log("It is the show() method from the child class");
11. }
12. }
13. var obj = **new** Child();
14. obj.show();

**Output**

It is the show() method from the parent class

It is the show() method from the child class