TypeScript is object oriented JavaScript. TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn’t support classes. Typescript gets this feature from ES6.

**Creating classes**

Use the class keyword to declare a class in TypeScript. The syntax for the same is given below −

**Syntax**

class class\_name {

//class scope

}

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

A class definition can include the following −

* **Fields** − A field is any variable declared in a class. Fields represent data pertaining to objects
* **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.

Consider a class Person in typescript.

class Person {

}

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Person = (function () {

function Person() {

}

return Person;

}());

**Example: Declaring a class**

class Car {

//field

engine:string;

//constructor

constructor(engine:string) {

this.engine = engine

}

//function

disp():void {

console.log("Engine is : "+this.engine)

}

}

The example declares a class Car. The class has a field named engine. The **var** keyword is not used while declaring a field. The example above declares a constructor for the class.

A constructor is a special function of the class that is responsible for initializing the variables of the class. TypeScript defines a constructor using the constructor keyword. A constructor is a function and hence can be parameterized.

The **this** keyword refers to the current instance of the class. Here, the parameter name and the name of the class’s field are the same. Hence to avoid ambiguity, the class’s field is prefixed with the **this** keyword.

*disp()* is a simple function definition. Note that the function keyword is not used here.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Car = (function () {

//constructor

function Car(engine) {

this.engine = engine;

}

//function

Car.prototype.disp = function () {

console.log("Engine is : " + this.engine);

};

return Car;

}());

**Creating Instance objects**

To create an instance of the class, use the **new** keyword followed by the class name. The syntax for the same is given below −

**Syntax**

var object\_name = new class\_name([ arguments ])

* 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 Car("Engine 1")

**Accessing Attributes and 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 an attribute

obj.field\_name

//accessing a function

obj.function\_name()

**Example: Putting them together**

class Car {

//field

engine:string;

//constructor

constructor(engine:string) {

this.engine = engine

}

//function

disp():void {

console.log("Function displays Engine is : "+this.engine)

}

}

//create an object

var obj = new Car("XXSY1")

//access the field

console.log("Reading attribute value Engine as : "+obj.engine)

//access the function

obj.disp()

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Car = (function () {

//constructor

function Car(engine) {

this.engine = engine;

}

//function

Car.prototype.disp = function () {

console.log("Function displays Engine is : " + this.engine);

};

return Car;

}());

//create an object

var obj = new Car("XXSY1");

//access the field

console.log("Reading attribute value Engine as : " + obj.engine);

//access the function

obj.disp();

The output of the above code is as follows −

Reading attribute value Engine as : XXSY1

Function displays Engine is : XXSY1

**Class Inheritance**

TypeScript supports the concept of Inheritance. Inheritance is the ability of a program to create new classes from an existing 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 private members and constructors from the parent class.

**Syntax**

class child\_class\_name extends parent\_class\_name

However, TypeScript doesn’t support multiple inheritance.

**Example: Class Inheritance**

class Shape {

Area:number

constructor(a:number) {

this.Area = a

}

}

class Circle extends Shape {

disp():void {

console.log("Area of the circle: "+this.Area)

}

}

var obj = new Circle(223);

obj.disp()

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var \_\_extends = (this && this.\_\_extends) || function (d, b) {

for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];

function \_\_() { this.constructor = d; }

d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_());

};

var Shape = (function () {

function Shape(a) {

this.Area = a;

}

return Shape;

}());

var Circle = (function (\_super) {

\_\_extends(Circle, \_super);

function Circle() {

\_super.apply(this, arguments);

}

Circle.prototype.disp = function () {

console.log("Area of the circle: " + this.Area);

};

return Circle;

}(Shape));

var obj = new Circle(223);

obj.disp();

The output of the above code is as follows −

Area of the Circle: 223

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 Car gets an implicit access to its parent class attribute i.e. area.

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. TypeScript doesn’t support multiple inheritance.
* **Multi-level** − The following example shows how multi-level inheritance works.

**Example**

class Root {

str:string;

}

class Child extends Root {}

class Leaf extends Child {} //indirectly inherits from Root by virtue of inheritance

var obj = new Leaf();

obj.str ="hello"

console.log(obj.str)

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

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var \_\_extends = (this && this.\_\_extends) || function (d, b) {

for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];

function \_\_() { this.constructor = d; }

d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_());

};

var Root = (function () {

function Root() {

}

return Root;

}());

var Child = (function (\_super) {

\_\_extends(Child, \_super);

function Child() {

\_super.apply(this, arguments);

}

return Child;

}(Root));

var Leaf = (function (\_super) {

\_\_extends(Leaf, \_super);

function Leaf() {

\_super.apply(this, arguments);

}

return Leaf;

}(Child));

var obj = new Leaf();

obj.str = "hello";

console.log(obj.str);

Its output is as follows −

**Output**

hello

**TypeScript ─ Class inheritance and Method Overriding**

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

class PrinterClass {

doPrint():void {

console.log("doPrint() from Parent called…")

}

}

class StringPrinter extends PrinterClass {

doPrint():void {

super.doPrint()

console.log("doPrint() is printing a string…")

}

}

var obj = new StringPrinter()

obj.doPrint()

The super keyword is used to refer to the immediate parent of a class. The keyword can be used to refer to the super class version of a variable, property or method. Line 13 invokes the super class version of the doWork() function.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var \_\_extends = (this && this.\_\_extends) || function (d, b) {

for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];

function \_\_() { this.constructor = d; }

d.prototype = b === null ? Object.create(b) : (\_\_.prototype = b.prototype, new \_\_());

};

var PrinterClass = (function () {

function PrinterClass() {

}

PrinterClass.prototype.doPrint = function () {

console.log("doPrint() from Parent called…");

};

return PrinterClass;

}());

var StringPrinter = (function (\_super) {

\_\_extends(StringPrinter, \_super);

function StringPrinter() {

\_super.apply(this, arguments);

}

StringPrinter.prototype.doPrint = function () {

\_super.prototype.doPrint.call(this);

console.log("doPrint() is printing a string…");

};

return StringPrinter;

}(PrinterClass));

var obj = new StringPrinter();

obj.doPrint();

The output of the above code is as follows −

doPrint() from Parent called…

doPrint() is printing a string…

**The static Keyword**

The static keyword can be applied to the data members of a class. A static variable retains its values till the program finishes execution. Static members are referenced by the class name.

**Example**

class StaticMem {

static num:number;

static disp():void {

console.log("The value of num is"+ StaticMem.num)

}

}

StaticMem.num = 12 // initialize the static variable

StaticMem.disp() // invoke the static method

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var StaticMem = (function () {

function StaticMem() {

}

StaticMem.disp = function () {

console.log("The value of num is" + StaticMem.num);

};

return StaticMem;

}());

StaticMem.num = 12; // initialize the static variable

StaticMem.disp(); // invoke the static method

The output of the above code is as follows −

The value of num is 12

**The instanceof operator**

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

**Example**

class Person{ }

var obj = new Person()

var isPerson = obj instanceof Person;

console.log(" obj is an instance of Person " + isPerson);

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Person = (function () {

function Person() {

}

return Person;

}());

var obj = new Person();

var isPerson = obj instanceof Person;

console.log(" obj is an instance of Person " + isPerson);

The output of the above code is as follows −

obj is an instance of Person True

**Data Hiding**

A class can control the visibility of its data members to members of other classes. This capability is termed as Data Hiding or Encapsulation.

Object Orientation uses the concept of access modifiers or access specifiers to implement the concept of Encapsulation. The access specifiers/modifiers define the visibility of a class’s data members outside its defining class.

The access modifiers supported by TypeScript are −

|  |  |
| --- | --- |
| **S.No.** | **Access Specifier & Description** |
| 1. | **public**  A public data member has universal accessibility. Data members in a class are public by default. |
| 2. | **private**  Private data members are accessible only within the class that defines these members. If an external class member tries to access a private member, the compiler throws an error. |
| 3. | **protected**  A protected data member is accessible by the members within the same class as that of the former and also by the members of the child classes. |

**Example**

Let us now take an example to see how data hiding works −

class Encapsulate {

str:string = "hello"

private str2:string = "world"

}

var obj = new Encapsulate()

console.log(obj.str) //accessible

console.log(obj.str2) //compilation Error as str2 is private

The class has two string attributes, str1 and str2, which are public and private members respectively. The class is instantiated. The example returns a compile time error, as the private attribute str2 is accessed outside the class that declares it.

**Classes and Interfaces**

Classes can also implement interfaces.

interface ILoan {

interest:number

}

class AgriLoan implements ILoan {

interest:number

rebate:number

constructor(interest:number,rebate:number) {

this.interest = interest

this.rebate = rebate

}

}

var obj = new AgriLoan(10,1)

console.log("Interest is : "+obj.interest+" Rebate is : "+obj.rebate )

The class AgriLoan implements the interface Loan. Hence, it is now binding on the class to include the property **interest** as its member.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var AgriLoan = (function () {

function AgriLoan(interest, rebate) {

this.interest = interest;

this.rebate = rebate;

}

return AgriLoan;

}());

var obj = new AgriLoan(10, 1);

console.log("Interest is : " + obj.interest + " Rebate is : " + obj.rebate);

The output of the above code is as follows −