**TYPESCRIPT**

**Topics:**

* What is Typescript?
* Why Typescript?
* Benefits of Typescript
* Oops
* Class
* Properties
* Methods
* Interface
* Type
* passing Parameter with type
* Constructor
* Extends
* Private
* Protected
* Readonly
* Object Desctructing
* Array Destructuring
* Template Litrals

**Installing & compiling:**

**Installing TypeScript :** > npm install -g typescript

In your editor, type the following JavaScript code in greeter.ts:

**function** **greeter**(person) {

**return** "Hello, " + person;

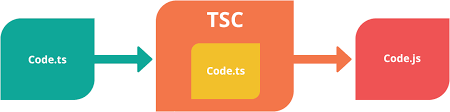
}

**let** user = "Jane User";

document.body.textContent = greeter(user);

## Compiling your code:

At the command line, run the TypeScript compiler: >tsc greeter.ts



**What is Typescript? (Transpiler)**

TypeScript is an object-oriented, open source programming language maintained and developed by Microsoft. It is developed to overcome the flaws of JavaScript.

**Why Typescript?**

As a superset of JavaScript, TypeScript can be used to develop large JavaScript applications with support for modules and classes, ES6 features, Type-checking, API definition, JavaScript packaging support, class library, and more.

* **Typescript** ES6 convert into ES5 features for browser compatability.
* **TypeScript** simplifies JavaScript code, making it easier to read and debug.
* **TypeScript** gives us all the benefits of ES6 (ECMAScript 6), plus more productivity.

**Difference between Typescript & Javascript:**

| **S.no** | **TypeScript** | **JavaScript** |
| --- | --- | --- |
| 1. | Typescript is an object-oriented language. | JavaScript is a scripting language. |
| 2. | Static typing feature is available here. | Doesn’t support static typing. |
| 3. | TypeScript supports modules. | JavaScript doesn’t support modules. |
| 4. | It has an interface. | Doesn’t have any interface. |
| 5. | The optional parameter function is supported here. | JavaScript doesn’t support optional parameter function. |

**Benefits of Typescript?**

Object-oriented — You got it right. TypeScript is Object-oriented. Classes, interfaces, inheritance …

Compilation — TypeScript is going to spit out any compilation error unlike the JavaScript counterpart which is an interpreted language. Programmers will be able to catch errors at the early stage during development. Thanks to the TypeScript transpiler!

EcmaScript6 — ES6 supports (JS advance features).

Strong\_typing — TypeScript is strongly-typed or supports static typing. You will not find this in JavaScript, which is a loosely-typed language. TypeScript also supports type inference using TLS (TypeScript Language Service).

Support for type definitions — TypeScript support type definitions for existing JavaScript libraries. You write type definitions in .d.ts file extension.

Used by Biggies— Yes. TypeScript is used by big companies and also by popular JavaScript frameworks. Angular 2, one of the popular JavaScript Framework is rewritten using TypeScript. In fact, Google Trends comparison shows that TypeScript has more internet points than competing compile-to-JavaScript frameworks.

**CLASS:**

Let’s take a look at a simple class-based example:

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

greet() {

**return** "Hello, " + **this**.greeting;

}

}

**let** greeter = **new** Greeter("world");

We declare a new class Greeter. This class has three members: a property called greeting, a constructor, and a method greet. You’ll notice that in the class when we refer to one of the members of the class we prepend this.. This denotes that it’s a member access. In the last line we construct an instance of the Greeter class using new. This calls into the constructor we defined earlier, creating a new object with the Greeter shape, and running the constructor to initialize it.

**Type**:

**Type** does not have a functionality of extending **Type** is mainly used when a union or tuple **type** needs to be used.

**type MoveDirection = ‘left’|’right’|’forward’|’reverse’;**

**class** car{

move(direction:MoveDirection){

}

}

let mycar = new car();

mycar.move(‘left’);

**INTERFACE:**

In **TypeScript**, an **interface** can create the new name that can be used everywhere. An **interface** can extend multiple **interfaces** and class as well.

**interface CarConfiguration {**

**model : Number,**

**brand : String**

**}**

**class** car{

constructor(carConfig:CarConfiguration){

this.model = carConfig.color;

this.brand = carConfig.brand;

}

}

let mycar = new car({

model : “v8”,

brand : “audi”

});

**Template Literals:** is nothing but string concatenation, in typescript we use the format as mentioned below

let name = ‘Ramya’

let salary = 1000;

In Javascript:

-------------------

var result = “Helo” +name+ “your salary is: “+salary+ “ ”; //Normal concatenation

o/p: Helo Ramya your salary is 1000

In Typescript: (called as Template Literals)

------------------

Var result = `Helo ${ name } , your salary is ${ salary }`; //backtic symbl (see ~ button)

o/p: Helo Ramya your salary is 1000

**Object Destructuring:**

In Javascript:

---------------------

var data = 5;

var result = { data: data} // same key and value means

var result = { id:1, data: data }

In Typescript:

-------------------

var data = 5;

var result = { data } // instead of 2 times using we can use one time in typescript

var result = { id:1, data }

**Array Destructuring:**

In Javascript:

---------------------

var \_a =[2,3], a = \_a[0] , b =\_a[1]

console.log(a);

In Typescript:

-------------------

var [a, b] = [2, 3] // directly assigning here

for eg., a = 2, b= 3

console.log(a) // 2 is o/p

console.log(b) // 3 is o/p

**Access Modifiers:** **TypeScript** supports three access modifiers - **public**, **private, and protected**.

**PRIVATE** **Method**:

When a member is marked **private**, it cannot be accessed from outside of its containing class.

**PROTECTED** **Method**:

The **protected** modifier acts much like the **private** modifier with the exception that members declared **protected** can also be accessed within deriving classes.

There is **one more** point to add regarding **Protected variables**:

when a base class variable is **protected** we cannot use its variable from **derived** class directly.

class car{

protected name: String;

constructor( name: string ) { this.name = name; }

}

class Mercedes extends car{

private noOfWheels: number;

constructor(name: string, noOfWheels: number ){

super(name);

this.noOfWheels = noOfWheels;

}

public getIntro(){

return “helo”;

}

}

let mycar = new Mercedes(“cool car” , 4);

console.log( mycar.getIntro() ); // helo

console.log( mycar.name ); // error // property ‘name’ is protected only accessible within class ‘car’ and its subclasses.

We can’t use variable **name** directly from outside of Car class, we can still use it from within an instance method of Mercedes because Mercedes derives from Car.

ReadOnly Modifier:

In TypeScript 2.0, the readonly modifier was added to the language. Properties marked with readonly can only be assigned to during initialization or from within a constructor of the same class. All other assignments are disallowed.

Let's take a look at an example. Here's a simple Point type that declares two read-only properties, x and y:

type Point = {

readonly x: number;

readonly y: number;

};

**OOPS:** There are four main principles to **Object Oriented** Programming: Encapsulation, Inheritance, Abstraction, and Polymorphism.

ENCAPSULATION:

Encapsulation is a key part of Object Oriented Programming that is a way to structure code so that a certain block of code has specific access points for external code. The term for this is “visibility” or “accessibility”. Visibility defines what code from one method, property, or class can call code in another method, property, or class.

Real-time example:

In other words, if we were to interact with a real world object like a car, we do so by using only certain parts of the car that are there for that purpose. These components of the car communicate to parts inside the car that perform the actions on our behalf. In the case of a car, these external parts are the pedals, steering, and dashboard controls, while the internal ones are in the engine. This means we don’t need to know what goes on under the car’s hood to make the car go, stop, or turn, all we need to do is interact with the correct component, called an interface. (i.e., the pedals, steering, dashboard controls). To enforce encapsulation in many languages, we use methods and properties. Both are blocks of code, although they differ mostly in syntax rather than conceptually. Behind each method or property is logic that often hides private data and performs complex tasks. The calling code shouldn’t and doesn’t need to know about those things.

In TypeScript, we enforce encapsulation with methods and properties that only allow access to data that we control.

The Withdraw method below does that by doing the calculation and updating the class level \_balance field. The Balance property then returns the private \_balance field to the calling code.

Withdraw(amount: number): boolean  
{  
    if (this.\_balance > amount)  
    {  
        this.\_balance -= amount  
        return true;  
    }  
    return false;  
}

private \_balance: number;  
get Balance(): number {  
    return this.\_balance;  
}

This encapsulates the account’s balance as business logic and validation code can run and access the private variables to formulate something to return to the client. In turn, the client never needs to know what goes on behind the scenes, just how to interact with the object. TypeScript’s syntax is similar to that of a statically typed language like C#, by using constructs like setters and getters.

INHERITANCE:

It’s quite easy to create an object model and inheritance chain with TypeScript. Just start with the familiar class keyword to create classes. The extends keyword causes the child class to inherit from the denoted base class.

For example, we can build out a very simplistic SavingsAccount and CheckingAccount classes that derive from a BankAccount class, like so:

module Bank {  
    class BankAccount { }

class SavingsAccount extends BankAccount { }

class CheckingAccount extends BankAccount { }  
}

## The two above TypeScript samples generate the JavaScript below. It’s very clear in TypeScript sample above that inheritance is happening, but not as clear in straight up JavaScript sample below. Sure, you could write this yourself, but as you can see, TypeScript generates the code you’re supposed to write, but without all the typing involved. There are only three lines of code needed for Bank class, but thirty for JavaScript. That’s a difference of ten times as much.

ABSTRACTION:

There is no formal mechanism for creating an abstract class in TypeScript, although a [request to implement abstraction in TypeScript has been issued](http://typescript.codeplex.com/workitem/395). Not to worry, for now we can use interfaces as a way to abstract class members. For example, a bank might have an Fee interface  
(their favorite one!) or an Interest interface that all account objects must implement. The Fee interface has members such as ChargeFee and the Interest interface contains one named CalculateInterest. When classes implement these interfaces, they must implement all of the members of the interface, as interfaces serve as digital contracts that classes can adhere to.

You can use the implements keyword to implement an interface in TypeScript, as shown here:

export module Bank  
{

export interface Fee { ChargeFee(amount: number); }

export class BankAccount implements Fee {  
        ChargeFee(amount: number) { }  
    }

}

POLYMORPHISM:

TypeScript enables polymorphism via method overrides as you can see in the example below. The Withdraw and Deposit methods of the CheckingAccount and SavingsAccount classes derive from the parent BankAccount class. In the child classes, we can override these methods and add our own business logic customizations, such as waving a fee if the balance is more than a specified amount.

class SavingsAccount extends BankAccount  
{

ChargeFee(amount: number)  
    {

if (this.Balance > 1000) { amount = 0; }  
         else { amount += 1.00; }

this.Balance =+ amount;

}

}

in this case, we overrided the ChargeFee that was originally part of the Fee interface. Polymorphism is an essential characteristic of OO programming, as code would be quite redundant without it.

## For more reference:

## <https://www.typescriptlang.org/docs/home.html>