TypeScript Note

Installation of TypeScript compiler (globally)

```
npm install -g typescript
```

Or in project only

```
npm install --save-dev typescript
```

Check Installation

```
tsc --version
```

tsconfig.json

```
tsc --init
```

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true
  }
}
```

In Visual Studio

After Creating a typescript file, visual studio shows a notification about installing a nuget package in the project. You can also manually install this package later. The package name is : Microsoft.TypeScript.MSBuild

First Code (Hello World)

helloworld.ts

```
let message: string = 'Hello World';
console.log(message);
```

Compile & Run

```
tsc helloworld.ts
node helloworld.js
```

Set output directory path (tsconfig.json)

```
{
    "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "sourceMap": true,
      "outDir": "out"
    }
  }
```

Language Features

Boolean

```
let isDone: boolean = false;
```

Number

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
```

```
let octal: number = 0o744;
```

String

```
let color: string = "blue";
color = "red";
```

Array

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

Tuple

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
Type 'number' is not assignable to type 'string'.
Type 'string' is not assignable to type 'number'.
console.log(x[0].substring(1)); // OK
console.log(x[1].substring(1)); // Error, 'number' does not have
'substring'
x[3] = "world"; // Error, Property '3' does not exist on type '[string,
number]'.
console.log(x[5].toString()); // Error, Property '5' does not exist on type
'[string, number]'.
```

Enum

```
enum Color {
```

```
  Red,
  Green,
  Blue
}
let c: Color = Color.Green;
```

```
enum Color
  Red = 1,
  Green,
  Blue
}
let c: Color = Color.Green;

enum Color
  Red = 1,
  Green = 2,
  Blue = 4
}
let c: Color = Color.Green;

enum Color
  Red = 1,
  Green,
  Blue
}
let colorName: string = Color[2];

console.log(colorName); // Displays 'Green' as its value is 2 above
```

Any

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
```

```
prettySure.toFixed(); // Error: method 'toFixed' doesn't exist on type
'Object'.
Property 'toFixed' does not exist on type 'Object'.
```

```
let list: any[] = [1, true, "free"];
list[1] = 100;
```

Void

```
function warnUser(): void {
  console.log("This is my warning message");
}
let unusable: void = undefined;
unusable = null; // OK if `--strictNullChecks` is not given
```

Null and Undefined

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

Never

```
// Function returning never must have unreachable end point
function error(message: string): never {
  throw new Error(message)
}

// Inferred return type is never
function fail() {
  return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
  while (true) {}
}
```

Object

```
declare function create(o: object | null): void;

create({ prop:0 }); // OK
create(null); // OK

create(42); // Error
Argument of type '42' is not assignable to parameter of type 'object |
null'.
create("string"); // Error
Argument of type '"string"' is not assignable to parameter of type 'object
| null'.
create(false); // Error
Argument of type 'false' is not assignable to parameter of type 'object |
null'.
create(undefined); // Error
Argument of type 'undefined' is not assignable to parameter of type 'object
| null'.
```

Type assertions

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

Classes

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
```

```
}

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

Interface

```
function printLabel(labeledObj: { label: string }) {
  console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

```
interface LabeledValue {
  label: string;
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

```
interface IPerson {
    name: string;
    gender: string;
}

interface IEmployee extends IPerson {
    empCode: number;
}

let empObj:IEmployee = {
    empCode:1,
    name:"Bill",
    gender:"Male"
}
```

```
interface IEmployee {
    empCode: number;
    name: string;
    getSalary:(number)=>number;
}

class Employee implements IEmployee {
    empCode: number;
    name: string;

    constructor(code: number, name: string) {
                this.empCode = code;
                this.name = name;
    }

    getSalary(empCode:number):number {
        return 20000;
    }
}

let emp = new Employee(1, "Steve");
```

Inheritance

```
class Animal {
  move(distanceInMeters: number = 0) {
    console.log(`Animal moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

```
class Animal {
  name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

Accessibility

```
class Animal {
  public name: string;
  public constructor(theName: string) {
    this.name = theName;
  }
  public move(distanceInMeters: number) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
```

```
class Animal {
  private name: string;
  constructor(theName: string) {
    this.name = theName;
  }
}

new Animal("Cat").name; // Error: 'name' is private;
```

```
class Animal {
  private name: string;
  constructor(theName: string) {
    this.name = theName;
  }
}

class Rhino extends Animal {
  constructor() {
    super("Rhino");
  }
}

class Employee {
  private name: string;
  constructor(theName: string) {
    this.name = theName;
  }
}
```

```
let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: 'Animal' and 'Employee' are not compatible
```

```
class Person {
  protected name: string;
  constructor(name: string) {
    this.name = name;
  }
}

class Employee extends Person {
  private department: string;

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

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in
${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

```
class Person {
  protected name: string;
  protected constructor(theName: string) {
    this.name = theName;
  }
}

// Employee can extend Person
```

```
class Employee extends Person {
  private department: string;

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

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in
${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is
protected
```

Optional Properties

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number
} {
  let newSquare = { color: "white", area: 100 };
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({ color: "black" });
```

Readonly

```
class Octopus {
  readonly name: string;
  readonly numberOfLegs: number = 8;
  constructor(theName: string) {
    this.name = theName;
  }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Indexable Types

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

```
class Animal {
  name: string;
}
class Dog extends Animal {
  breed: string;
}

// Error: indexing with a numeric string might get you a completely
separate type of Animal!
interface NotOkay {
  [x: number]: Animal;
  [x: string]: Dog;
}
```

```
interface NumberDictionary {
  [index: string]: number;
```

```
  length: number; // ok, length is a number
  name: string; // error, the type of 'name' is not a subtype of the
indexer
}
```

```
interface NumberOrStringDictionary {
  [index: string]: number | string;
  length: number; // ok, length is a number
  name: string; // ok, name is a string
}
```

```
interface ReadonlyStringArray {
  readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

Parameter properties

```
class Octopus {
  readonly numberOfLegs: number = 8;
  constructor(readonly name: string) {}
}
```

Accessors

```
class Employee {
  fullName: string;
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
  console.log(employee.fullName);
}
```

```
const fullNameMaxLength = 10;

class Employee {
```

```
  private _fullName: string;

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
    if (newName && newName.length > fullNameMaxLength) {
      throw new Error("fullName has a max length of " + fullNameMaxLength);
    }

    this._fullName = newName;
  }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
  console.log(employee.fullName);
}
```

Static Properties

```
class Grid {
  static origin = { x: 0, y: 0 };
  calculateDistanceFromOrigin(point: { x: number; y: number }) {
    let xDist = point.x - Grid.origin.x;
    let yDist = point.y - Grid.origin.y;
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
  constructor(public scale: number) {}
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({ x: 10, y: 10 }));
console.log(grid2.calculateDistanceFromOrigin({ x: 10, y: 10 }));
```

```
class DemoCounter {
    static counter : number = 0;
    name : string = "";
    increament(name: string) : void {
        DemoCounter.counter++;
        this.name = name;
    }
    static doSoemthing(): void{
        console.log("printing from static method");
    }
}

let demoCounter : DemoCounter = new DemoCounter();
let demoCounter2 : DemoCounter = new DemoCounter();
demoCounter.increament("x");
demoCounter.increament("y");
demoCounter2.increament("z");
console.log(DemoCounter.counter);
console.log(demoCounter.name);
console.log(demoCounter2.name);
DemoCounter.doSoemthing();
```

Abstract Classes

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log("roaming the earth...");
  }
}
```

```
abstract class Department {
  constructor(public name: string) {}
```

```
  printName(): void {
    console.log("Department name: " + this.name);
  }

  abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {
  constructor() {
    super("Accounting and Auditing"); // constructors in derived classes
must call super()
  }

  printMeeting(): void {
    console.log("The Accounting Department meets each Monday at 10am.");
  }

  generateReports(): void {
    console.log("Generating accounting reports...");
  }
}

let department: Department; // ok to create a reference to an abstract type
department = new Department(); // error: cannot create an instance of an
abstract class
department = new AccountingDepartment(); // ok to create and assign a
non-abstract subclass
department.printName();
department.printMeeting();
department.generateReports(); // error: method doesn't exist on declared
abstract type
```

Constructor functions

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
```

```
  }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet()); // "Hello, world"
```

```
let Greeter = (function() {
  function Greeter(message) {
    this.greeting = message;
  }
  Greeter.prototype.greet = function() {
    return "Hello, " + this.greeting;
  };
  return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet()); // "Hello, world"
```

Using a class as an interface

```
class Point {
  x: number;
  y: number;
}

interface Point3d extends Point {
  z: number;
}

let point3d: Point3d = { x: 1, y: 2, z: 3 };
```

Generics

```
function identity(arg: number): number {
  return arg;
}
```

```
function identity(arg: any): any {
```

```
  return arg;
}
```

```
function identity<T>(arg: T): T {
  return arg;
}
```

```
let output = identity<string>("myString");
```

Namespaces

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }

  const lettersRegexp = /^[A-Za-z]+$/;
  const numberRegexp = /^[0-9]+$/;

  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegexp.test(s);
    }
  }

  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegexp.test(s);
    }
  }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();
```

```
// Show whether each string passed each validator
for (let s of strings) {
  for (let name in validators) {
    console.log(
      `"${s}" - ${
        validators[name].isAcceptable(s) ? "matches" : "does not match"
      } ${name}`
    );
  }
}
```

Multi file Namespace

Validation.ts

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
  const lettersRegexp = /^[A-Za-z]+$/;
  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegexp.test(s);
    }
  }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];
```

```
// Validators to use
let validators: { [s: string]: Validation.StringValidator } = {};
validators["Letters only"] = new Validation.LettersOnlyValidator();
```

Aliases

```
namespace Shapes {
  export namespace Polygons {
    export class Triangle {}
    export class Square {}
  }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Decorators

Command Line:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

```
function f() {
  console.log("f(): evaluated");
  return function(target, propertyKey: string, descriptor:
PropertyDescriptor) {
    console.log("f(): called");
  };
}
```

```
function g() {
  console.log("g(): evaluated");
  return function(target, propertyKey: string, descriptor:
PropertyDescriptor) {
    console.log("g(): called");
  };
}

class C {
  @f()
  @g()
  method() {}
}
```

Triple-Slash Directives

```
/// <reference path="..." />
/// <reference types="..." />
/// <reference lib="..." />
/// <reference no-default-lib="true"/>
/// <amd-module />
/// <amd-dependency />
```

Union Types

```
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

```
// passes at compile time, fails at runtime.
let indentedString = padLeft("Hello world", true);
```

```
function padLeft(value: string, padding: string | number) {
  // ...
}

let indentedString = padLeft("Hello world", true); // Compile error
```

Type Aliases

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
  if (typeof n === "string") {
    return n;
  } else {
    return n();
  }
}
```

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
```

Intersection Types

```
interface ErrorHandling {
  success: boolean;
  error?: { message: string };
}

interface ArtworksData {
  artworks: { title: string }[];
}

interface ArtistsData {
  artists: { name: string }[];
```

```
}

type ArtworksResponse = ArtworksData & ErrorHandling;
type ArtistsResponse = ArtistsData & ErrorHandling;

const handleArtistsResponse = (response: ArtistsResponse) => {
  if (response.error) {
    console.error(response.error.message);
    return;
  }

  console.log(response.artists);
};
```