TypeScript

LUXOFT

LXFT
LISTED
NYSE

TypeScript

TypeScript is a free and open-source programming language developed and maintained by Microsoft. It is a strict superset of JavaScript, and adds optional static typing.

TypeScript is designed for development of large applications and transcompiles to JavaScript.

As TypeScript is a superset of JavaScript, any existing JavaScript programs are also valid TypeScript programs.

***Anders Hejlsberg**, lead architect of C#*
creator of Delphi and Turbo Pascal,
author of TypeScript

Data types

Boolean
```
isDone: boolean = false;
```

Number:
```
height: number = 6;
```

String:
```
name: string = "bob";
```

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

Enum:
```
enum Color {Red, Green, Blue};
c: Color = Color.Green;
```

Any:
```
notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
var list:any[] = [1, true, "free"];
```

Void:
```
function warnUser(): void {
    alert("This is my warning message");
}
```

Tuples

```
// Declare a tuple type
let x: [string, number];
x = ["hello", 10]; // OK
x = [10, "hello"]; // Error

console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'

// Return tuple from function
function f(): [string, number] {
   return [“cow”,3];
}
```

Type 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) {
    }
}
```

Type assertions

A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data.

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

And the other is the as-syntax:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as
string).length;
```

Type aliases

```
type PrimitiveArray = Array<string|number|boolean>;
type MyNumber = number;
type Callback = () => void;

let f: Callback;
f = function() {
    console.log(“function");
}
```

Interfaces

Define right in place:

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

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

Using interface keyword:

```
interface LabelledValue {
    label: string;
}
function printLabel(labelledObj: LabelledValue) {
console.log(labelledObj.label);
}
var myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Interfaces: optional properties

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

function createSquare(config: SquareConfig):
    {color: string; area: number} {
    var newSquare = {color: "white", area: 100};
    if (config.color) {
        newSquare.color = config.color;
        // Type-checker can catch the mistyped name
here
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

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

Interfaces: function types

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

var mySearch: SearchFunc;
mySearch = function(source: string, subStr: string) {
    var result = source.search(subStr);
    if (result == -1) {
        return false;
    } else {
        return true;
    }
}
```

Interfaces: array types

```
interface StringArray {
    [index: number]: string;
}
var myArray: StringArray;
myArray = ["Bob", "Fred"];
```

Interfaces: class types

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;

    setTime(d: Date) {
        this.currentTime = d;
    }

    constructor(h: number, m: number) { }
}
```

Interfaces: static/instance side of class

```
interface ClockStatic {
    new (hour: number, minute: number);
}

class Clock {
    currentTime: Date;

    constructor(h: number, m: number) { }
}

var cs: ClockStatic = Clock;
var newClock = new cs(7, 30);

class Timer {
    constructor(h: number, m: number) { }
}
cs = Timer;
var newTimer = new cs(7, 30);
```

Extending Interfaces

```
interface Shape {
	color: string;
}

interface PenStroke {
	penWidth: number;
}

interface Square extends Shape, PenStroke {
	sideLength: number;
}

var square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

Interfaces: Hybrid Types

```
interface Counter {
        (start: number): string;
        interval: number;
        reset(): void;
}

var c: Counter;
c(10);
c.reset();
c.interval = 5.0;
```

Classes

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
        return "Hello, " + this.greeting;
    }
}

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

Private/Public/Protected: Public by default

```
class Animal {
    private name:string;

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

    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Parameter properties:

```
class Animal {
    constructor(private name: string) { }
    move(meters: number) {
        alert(this.name + " moved " + meters + "m.");
    }
}
```

Accessors

```
var passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string { return this._fullName; }
    set fullName(newName: string) {
        if (passcode && passcode == "secret passcode") {
            this._fullName = newName;
        } else {
            alert("Error: Unauthorized update!");
        }
    }
}

var employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

Static properties

```
class Grid {
    static origin = {x: 0, y: 0};

    calculateDistanceFromOrigin(point: {x: number; y: number;}) {
        var xDist = (point.x - Grid.origin.x);
        var yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }

    constructor (public scale: number) { }
}

var grid1 = new Grid(1.0); // 1x scale
var grid2 = new Grid(5.0); // 5x scale
alert(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
alert(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

Constructor function

```
class Greeter {
    static standardGreeting = "Hello, there";
    greeting: string;

    greet() {
        if (this.greeting) { return "Hello, " + this.greeting; }
        else { return Greeter.standardGreeting; }
    }
}

var greeter1: Greeter;
greeter1 = new Greeter();
alert(greeter1.greet());

var greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";
var greeter2:Greeter = new greeterMaker();
alert(greeter2.greet());
```

Using a class as an interface

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

interface Point3d extends Point {
        z: number;
}

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

Functions

```
function add(x: number, y: number): number { return x+y; }

var myAdd = function(x: number, y: number): number { return x+y; };
```

Writing the function type:

```
var myAdd: (a:number, b:number)=>number =
    function(x: number, y: number): number { return x+y; };
```

Inferring the types:

```
// The parameters 'x' and 'y' have the type number
var myAdd: (baseValue:number, increment:number)=>
    number = function(x, y) { return x+y; };
```

Functions

Optional parameters:

```
    function buildName(firstName: string, lastName?: string) {
        if (lastName) return firstName + " " + lastName;
        else return firstName;
    }

    var result1 = buildName("Bob"); //works correctly now
    var result2 = buildName("Bob", "Adams", "Sr."); //error, too many params
    var result3 = buildName("Bob", "Adams"); //ah, just right
```

Default parameters:

```
    function buildName(firstName: string, lastName = "Smith") {
        return firstName + " " + lastName;
    }

    var result1 = buildName("Bob"); //works correctly now, also
    var result2 = buildName("Bob", "Adams", "Sr."); //error, too many
params
    var result3 = buildName("Bob", "Adams"); //ah, just right
```

Functions

Rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}
var employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Functions overloading

```
var suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    // Check to see if we're working with an object/array
    if (typeof x == "object") {
        var pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    } // Otherwise just let them pick the card
    else if (typeof x == "number") {
        var pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}

var myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }];
var pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
var pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Generics

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

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

Using generics:

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

Pass type in <>:

```
    var output = identity<string>("myString"); // type of output will be 'string‘
```

Interfere type automatically:

```
    var output = identity("myString"); // type of output will be 'string'
```

Generics

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length); // Error: T doesn't have .length
    return arg;
}
```

We can define that we are using array:

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length); // Array has a .length, so no more error
    return arg;
}
```

Alternatively:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);
    // Array has a .length, so no more error
    return arg;
}
```

Generic types:

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

var myIdentity: <T>(arg: T)=>T = identity;
```

Generic Classes

```
class GenericNumber<T> {
        zeroValue: T;
        add: (x: T, y: T) => T;
}

var myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };



var stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

alert(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Generic constraints

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length); // Error: T doesn't have .length
    return arg;
}
```

Solution using constraint:

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);
    // Now we know it has a .length property, so no more error
    return arg;
}

loggingIdentity(3); // Error, number doesn't have a .length property
loggingIdentity({length: 10, value: 3}); // OK
```

Using class type in generics

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

Example of using:

```
class BeeKeeper { hasMask: boolean; }
class ZooKeeper { nametag: string; }
class Animal { numLegs: number; }
class Bee extends Animal { keeper: BeeKeeper; }
class Lion extends Animal { keeper: ZooKeeper; }

function findKeeper<A extends Animal, K> (a: {new(): A;
    prototype: {keeper: K}}): K {
        return a.prototype.keeper;
}

findKeeper(Lion).nametag; // typechecks!
```

Merging interfaces

```
interface Box {
	height: number;
	width: number;
}

interface Box { scale: number; }

var box: Box = {height: 5, width: 6, scale: 10};
```

Type Inference

- basic:

  ```
  x = 3 // inferred to number
  ```

- best common type:

  ```
  var x = [0, 1, null];
  ```

To infer the type of x in the example above, we must consider the type of each array element. Here we are given two choices for the type of the array: **number** and **null**. The best common type algorithm considers each candidate type, and picks the type that is compatible with all the other candidates.

- types share a common structure, but no one is the super type of all candidate types:

  ```
          var zoo = [new Rhino(), new Elephant(), new Snake()];
  ```

Ideally, we may want zoo to be inferred as an Animal[], but because there is no object that is strictly of type Animal - to correct use :

```
var zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

Contextual type

Type of an expression is implied by its location

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.buton); //<- Error
};
```

For the code above to give the type error, the TypeScript type checker used the type of the Window.onmousedown function to infer the type of the function expression on the right hand side of the assignment.

Solution:

```
window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.button); //<- Now, no error is given
};
```

explicit type override the contextual type:

```
    function createZoo(): Animal[] {
        return [new Rhino(), new Elephant(), new Snake()];
    }
```

Type Compatibility

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members.

```
interface Named { name: string; }
class Person {
    name: string;
}
var p: Named; // OK, because of structural typing
p = new Person();
```

x is compatible with y if y has at least the same members as x:

```
interface Named {  name: string; }
var x: Named; // y's inferred type is { name: string; location: string; }
var y = { name: 'Alice', location: 'Seattle' };
x = y; // OK!
```

the same for checking function call arguments:

```
function greet(n: Named) {
    alert('Hello, ' + n.name);
}
greet(y); // OK
```

Decorators

```
class C {
    @readonly
    @enumerable(false)
    method() { }
}

function readonly(target, key, descriptor) {
    descriptor.writable = false;
}

function enumerable(value) {
    return function (target, key, descriptor) {
        descriptor.enumerable = value;
    }
}
```

Class expressions (anonymous class type)

```
let Point = class {
    constructor(public x: number, public y: number) { }
    public length() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
};
var p = new Point(3, 4);  // p has anonymous class type
console.log(p.length());
```

Extending expressions

```
// Extend built-in types
class MyArray extends Array<number> { }
class MyError extends Error { }

// Extend computed base class
class ThingA {   getGreeting() { return "Hello from A"; }  }
class ThingB {   getGreeting() { return "Hello from B"; }  }
interface Greeter {   getGreeting(): string;   }
interface GreeterConstructor {    new (): Greeter;   }

function getGreeterBase(): GreeterConstructor {
    return Math.random() >= 0.5 ? ThingA : ThingB;
}
class Test extends getGreeterBase() {
    sayHello() {
        console.log(this.getGreeting());
    }
}
```

Abstract classes

```
abstract class Base {
    abstract getThing(): string;
    getOtherThing() { return 'hello'; }
}
let x = new Base(); // Error, 'Base' is abstract


class Derived extends Base {
    getThing() { return 'hello'; }
}

var x = new Derived(); // OK
var y: Base = new Derived(); // Also OK
y.getThing(); // OK
y.getOtherThing(); // OK
```

Async/await

```
// printDelayed is a 'Promise<void>'
async function printDelayed(elements: string[]) {
    for (const element of elements) {
        await delay(200);
        console.log(element);
    }
}

async function delay(milliseconds: number) {
    return new Promise<void>(resolve => {
        setTimeout(resolve, milliseconds);
    });
}

printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {
    console.log();
    console.log("Printed every element!");
});
```