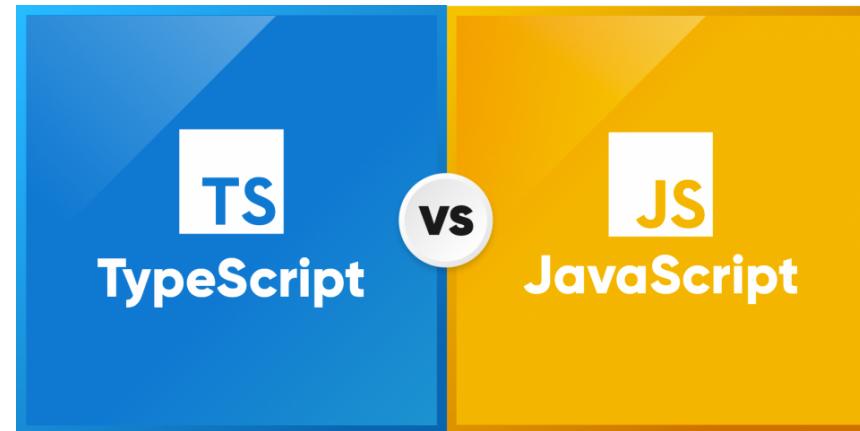


Making life easier with Typescript



Some slides adapted from LifeMichael.com

Polyfilling and transpiling

- ❖ **Polyfilling** is one of the methodologies that can be used as a sort of backward compatibility measurement.
- ❖ “A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. (Remy Sharp).”
- ❖ a “**Transpiler**” is a tool that transforms code with newer syntax into older code equivalents. This process is called “**Transpiling**”.

TypeScript

- ❖ The TypeScript programming language was developed by Microsoft. It is an open source programming language.
- ❖ The code we write in TypeScript is compiled into JavaScript (**transpiled**)
- ❖ TypeScript gives us the capabilities, which are required to develop large scale applications using JavaScript.

TypeScript

- ❖ TypeScript is a superset of JavaScript. It includes the entire JavaScript programming language together with additional capabilities.
- ❖ TypeScript allows us to use JavaScript as if it was a strictly type programming language.
 - ❖ TypeScript allows us to specify the type of the variables.
 - ❖ TypeScript allows us to define classes and interfaces.

TypeScript

- ❖ In general, nearly every code we can write in JavaScript can be included in code we write in TypeScript.
- ❖ Compiling TypeScript into JavaScript we get a clean simple ES3 compliant code we can run in any web browser

The TypeScript playground

<https://www.typescriptlang.org/play/>

TS Config ▾ Examples ▾ What's New ▾ Settings

v3.9.2 ▾ Run Export → JS DTS Errors Logs Plugins

```
1 class Greeting {  
2     greet():void {  
3         console.log("Hello World!!!!")  
4     }  
5 }  
6 var obj = new Greeting();  
7 obj.greet();
```

```
"use strict";  
class Greeting {  
    greet() {  
        console.log("Hello World!!!!");  
    }  
}  
var obj = new Greeting();  
obj.greet();
```

EcmaScript 2017



Configuring TypeScript playground

TS TypeScript Download Documentation Handbook Community Playground Tools

Playground TS Config ▾ Examples ▾ What's New ▾

Close

TS Config

Lang: `TypeScript`

Which language should be used in the editor

Target: `ES3`

Set the supported JavaScript language runtime to transpile to

JSX: `None`

Control how JSX is emitted

Module: `None`

Sets the expected module system for your runtime

Configuring TypeScript playground

TS Config ▾ Examples ▾ What's New ▾

Settings

v3.9.2 ▾ Run Export ▾

```
1 class Greeting {  
2     greet():void {  
3         console.log("Hello World!!!")  
4     }  
5 }  
6 var obj = new Greeting();  
7 obj.greet();
```

→

JS DTS Errors Logs Plugins

```
"use strict";  
  
var Greeting = /** @class */ (function () {  
    function Greeting() {}  
    Greeting.prototype.greet = function () {  
        console.log("Hello World!!!");  
    };  
    return Greeting;  
}());  
  
var obj = new Greeting();  
obj.greet();
```

ES 3

Configuring TypeScript playground

The screenshot shows the TypeScript playground interface. At the top, there's a navigation bar with a yellow oval containing a React logo and the word "React". Other menu items include "Docs", "Tutorial", "Blog", "Community", and a search icon. Below the navigation bar, the main content area has a large title "Introduzione a JSX". Underneath the title, there's a text block that says "Considera questa dichiarazione di variabile:" followed by a code snippet:

```
const element = <h1>Hello, world!</h1>;
```

. To the right of this, there's a configuration section with a dropdown labeled "JSX:" set to "None" and a subtitle "Control how JSX is emitted". A red arrow points from this configuration section towards the JSX code example. The background of the playground has a blue header with "Handbook", "Community", and "Playground" links.

Questa strana sintassi con i tag non è né una stringa né HTML.

È chiamata JSX, ed è un'estensione della sintassi JavaScript. Ti raccomandiamo di utilizzarla con React per descrivere l'aspetto che dovrebbe avere la UI (*User Interface*, o interfaccia utente). JSX ti potrebbe ricordare un linguaggio di template, ma usufruisce di tutta la potenza del JavaScript.

Variable typing

We define the type of the variables:

name:type

```
var id:number = 221255;  
var aname:string = "Dorothea";  
var tall:boolean = true;  
var names:string[] = ['pippo','pluto','minnie'];
```

JS 'use strict'

- It is forbidden to use undeclared variables:

```
x=3; // must become var x=3;
```

- any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.
- Deleting a variables, objects, functions is not allowed.
- function parameter names be unique. In normal code the last duplicated argument hides previous identically-named arguments: those previous arguments remain available through arguments[i].

For more, see

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Function typing

We define the type of the variables:

name:type

The screenshot shows the TypeScript playground interface. At the top, there's a navigation bar with the TypeScript logo, 'TypeScript' text, and links for 'Download', 'Docs', 'Handbook', 'Community', and 'Tools'. A search bar says 'Search Docs'. Below the navigation is a toolbar with 'Playground', 'TS Config ▾', 'Examples ▾', 'What's New ▾', and 'Settings'. The main area has dropdowns for 'v3.9.2 ▾', 'Run', and 'Export ▾'. On the right, there are tabs for 'JS', 'DTS', 'Errors', 'Logs', and 'Plugins', with 'JS' being the active tab. The left pane contains TypeScript code, and the right pane shows the generated JavaScript code.

```
1 function sum(a:number,b:number):number
2 {var temp:number = a+b; return temp; }
3
4 var result:number = sum(7,18);
5 document.write("result="+result);
6
7 function doSomething():void
8 { document.write("<h1>hola</h1>") }
9
10 doSomething();
11
```

```
"use strict";
function sum(a, b) { var temp = a + b; return temp; }
var result = sum(7, 18);
document.write("result=" + result);
function doSomething() { document.write("<h1>hola</h1>"); }
doSomething();
```

Number of params

- ❖ Unlike JavaScript, when calling a function passing over arguments the number of arguments must match the number of the parameters, otherwise then we get a compilation error.

The screenshot shows a TypeScript playground interface. At the top, there are navigation links: 'Playground', 'TS Config ▾', 'Examples ▾', 'What's New ▾', and 'Settings'. Below this is a toolbar with dropdowns for 'v3.9.2 ▾', 'Run', 'Export ▾', and a '→' button. On the right, there are tabs for 'JS', 'DTS', 'Errors', 'Logs', and 'Plugins', with 'JS' being the active tab. The code editor contains the following TypeScript code:

```
1 function sum(a:number,b:number):number
2 {var temp:number = a+b; return temp; }
3
4 var result:number = sum(7);
5 document.write("result="+result);
6
7
```

The code is compiled into the following JavaScript output in the 'JS' panel:

```
"use strict";
function sum(a, b) { var temp = a + b; return temp; }
var result = sum(7);
document.write("result=" + result);
```

Below the JS panel, the text 'Result: NaN' is displayed in red, indicating the execution result of the code.

Optional params

- ❖ Adding the question mark to the name of a parameter will turn that parameter into an optional one.
- ❖ The optional parameters should be after any other required one. They should be the last ones.

```
function sum(a:number,b:number,c?:number) :number
{
    var total = 0;
    if(c!==undefined) { total += c; }
    total += (a+b); return total;
}

var temp = sum(7,8);
document.write("temp="+temp);
```

Default params

- ❖ When defining a function we can specify default values for any of its parameters. Doing so, if an argument is not passed over to the parameter then the default value we specified will be set instead.

The screenshot shows a TypeScript playground interface. On the left, the code editor contains the following TypeScript code:

```
1 function dosomething(a:number,b:number,step:number=((b-a)%5)) {  
2     var i=a;  
3     while(a<=b) {  
4         document.write("<br/>" + a); a+=step;  
5     }  
6 }  
7  
8 dosomething(1,20);  
9
```

On the right, the results panel shows the transpiled JavaScript code:

```
"use strict";  
  
function dosomething(a, b, step) {  
    if (step === void 0) { step = ((b - a) % 5); }  
    var i = a;  
    while (a <= b) {  
        document.write("<br/>" + a);  
        a += step;  
    }  
}  
  
dosomething(1, 20);
```

Rest params

- ❖ We can define a function with an arbitrary number of params (like the "main" in Java).

The screenshot shows the TypeScript playground interface. At the top, there are navigation links: Playground, TS Config ▾, Examples ▾, What's New ▾, and Settings. Below that, there are dropdowns for version (v3.9.2), Run, Export, and a status bar with a right-pointing arrow. On the far right, there are tabs for JS, DTS, Errors, Logs, and Plugins, with JS selected.

The code editor contains the following TypeScript code:

```
1  function sum(...numbers: number[]):number
2  {
3  var total:number = 0;
4  for(var i=0; i<numbers.length; i++)
5  {
6  total += numbers[i];
7  }
8  return total;
9  }
10
11 document.write("<br/>" + sum(2,5,3));
12
13
```

The output pane shows the generated JavaScript code:

```
"use strict";
function sum() {
    var numbers = [];
    for (var _i = 0; _i < arguments.length; _i++)
        numbers[_i] = arguments[_i];
}
var total = 0;
for (var i = 0; i < numbers.length; i++) {
    total += numbers[i];
}
return total;
}
document.write("<br/>" + sum(2, 5, 3));
```

At the bottom left, there is a small logo with the letters 'AT'. At the bottom right, there is a logo for the University of Trento (University of Trento) with the text 'UNIVERSITÀ DI TRENTO'.

TypeScript

- ❖ When starting from standard JavaScript, you might get some errors due to the differences between JavaScript and TypeScript.
- ❖ For instance, you get an error if you treat a variable in our code as if it was a dynamic type variable (as in JavaScript).
- ❖ Unlike other programming languages, when getting error messages from the TypeScript compiler it will still try to execute the code.

treating a variable as if it was a dynamic type variable

TS Config ▾ Examples ▾ What's New ▾ Settings

v3.9.2 ▾ Run Export ▾

```
1 var myNumber:number;
2 myNumber="pippo";
```



JS DTS Errors Logs Plugins

```
"use strict";
var myNumber;
myNumber = "pippo";
```

TS Config ▾ Examples ▾ What's New ▾ Settings

v3.9.2 ▾ Run Export ▾



JS DTS Errors Logs Plugins

```
1 var myNumber:number;
2 var myNumber: number
```

Type ""pippo"" is not assignable to type
'number'. (2322)

Peek Problem No quick fixes available

Dynamic type variables

We can create a variable with a dynamic type if we specify its type to be any.

```
var temp:any = 3;  
temp = 'a';  
temp = [23,5,23];  
temp = true;  
temp = new Object();
```

Classes

```
class Car {  
    //field  
    engine:string;  
  
    //constructor  
    constructor(engine:string) {  
        this.engine = engine  
    }  
  
    //function  
    disp():void {  
        console.log("Engine is : "+this.engine)  
    }  
}
```

Constructor

- ❖ When we define a new class it automatically has a constructor, the default one.
- ❖ We can define a new constructor. When doing so, the default one will be deleted.
- ❖ There is no constructor polymorphism.
- ❖ When we define a new constructor we can specify each one of its parameters with an access modifier and by doing so indirectly define those parameters as instance variables

Access modifiers

- ❖ The available access modifiers are **private**, **public** and **protected**. The **public** access modifier is the default one. If we don't specify an access modifier then it is **public**.

Instance vars and methods

- ❖ The variables are usually declared before the constructor. Each variable definition includes three parts. The optional access modifier, the identifier and the type annotation.
- ❖ The methods are declared without using the function keyword. We can precede the function name with an access modifier and we can append the function declaration with the type of its returned value.

Instance vars and methods

```
class Rectangle
{
    private width:number;  private height:number;
    constructor(width:number, height:number)
    {
        this.width = width;
        this.height = height;
    }

    protected area():number
    {
        return this.width*this.height;
    }
}
```

Static vars and methods

- ❖ We can define static variables and static methods by adding the `static` keyword. Accessing static variables and methods is done using the name of the class.

Static vars and methods

```
class FinanceUtils
{
    public static VAT = 0.18;
    public static calculateTax(sum:number) :number
    {
        return 0.25*sum;
    }
    public static calculateVAT(sum:number) :number
    {
        return FinanceUtils.VAT*sum;
    }
}

var price:number = 1020;
document.write("<br/>"+FinanceUtils.calculateVAT(price)) ;
```

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()
```

Type assertion

- ❖ *Type assertions* are a way to tell the compiler “trust me, I know what I’m doing.” A type assertion is like a type cast in other languages, but performs no special checking. It has no runtime impact, and is used purely by the compiler.

```
class Person {  
    id:number;  
    name:string;  
}  
  
class Student extends Person  
{  
    average:number;  
}  
  
var a:Person = new Student();  
var b:Student = <Student>a;
```

Generics

```
// for number type
function fun(args: number):
number {
    return args;
}

// for string type
function fun(args: string):
string {
    return args;
}

let result = fun<string>("Hello
World");

let result2 = fun<number>(200);
```

```
function fun(args: any): any
{
    return args;
}

function fun<T>(args:T):T {
    return args;
}
```

Other class related issues

- ❖ TypeScript doesn't support multiple inheritance.

- ❖ `super`

- ❖ Classes implement interfaces

```
class Student implements Iprintable { ... }
```

The next big thing: interfaces

- ❖ We can use Interfaces as data type definition
- ❖ They fully disappear in JavaScript!

The screenshot shows the TypeScript playground interface. The top navigation bar includes 'Playground', 'TS Config ▾', 'Examples ▾', 'What's New ▾', and 'Settings'. Below the bar, there are dropdowns for 'v3.9.2 ▾', 'Run', 'Export ▾', and a build status icon. The main area has tabs for 'JS' (selected), 'DTS', 'Errors', 'Logs', and 'Plugins'. On the left, the code editor contains the following TypeScript code:

```
1  interface IPerson {  
2      firstName:string,  
3      lastName:string,  
4      sayHi: ()=>string  
5  }  
6  
7  var customer:IPerson = {  
8      firstName:"Tom",  
9      lastName:"Hanks",  
10     sayHi: ():string =>{return "Hi there"}  
11  }  
12  
13  console.log("Customer Object ")  
14  console.log(customer.firstName)  
15  console.log(customer.lastName)  
16  console.log(customer.sayHi())
```

The right panel displays the generated JavaScript code:

```
"use strict";  
  
var customer = {  
    firstName: "Tom",  
    lastName: "Hanks",  
    sayHi: function () { return "Hi there"; }  
};  
  
console.log("Customer Object ");  
console.log(customer.firstName);  
console.log(customer.lastName);  
console.log(customer.sayHi());
```

interfaces multiple inheritance

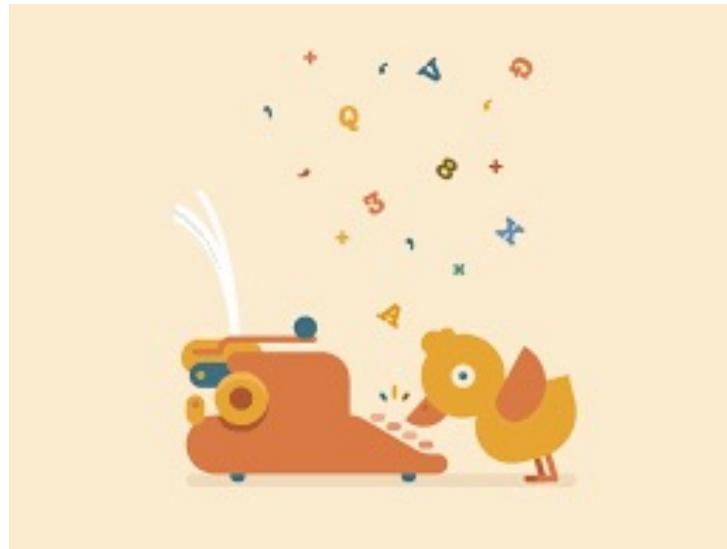
```
interface IParent1 {  
    v1:number  
}  
  
interface IParent2 {  
    v2:number  
}  
  
interface Child extends IParent1, IParent2 { }  
var Iobj:Child = { v1:12, v2:23}  
console.log("value 1: "+this.v1+ " value 2: "+this.v2)
```

duck inheritance

```
class Vehicle {  
    public run(): void { console.log('Vehicle.run') ; }  
}  
  
class Task {  
    public run(): void { console.log('Task.run') ; }  
}  
  
function runTask(t: Task) {  
    t.run();  
}  
  
runTask(new Task());  
runTask(new Vehicle());
```

Duck Typing

- ❖ Duck typing in computer programming is an application of the duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck"—to determine if an object can be used for a particular purpose. With normal typing, suitability is determined by an object's type.



avoiding duck inheritance - 1

```
class Vehicle {  
    private x: string="A";  
    public run(): void { console.log('Vehicle.run'); }  
}  
  
class Task {  
    private x: string="A";  
    public run(): void { console.log('Task.run'); }  
}  
  
function runTask(t: Task) {  
    t.run();  
}  
  
runTask(new Task());  
runTask(new Vehicle()); // Will be a compile time error
```

Argument of type 'Vehicle' is not assignable to parameter of type 'Task'.

Types have separate declarations of a private property 'x'.(2345)

avoiding duck inheritance - 2

```
class Vehicle {  
    private x: string="A";  
    public run(): void { console.log('Vehicle.run'); }  
}  
  
class Task {  
    private s: string="A";  
    public run(): void { console.log('Task.run'); }  
}  
  
function runTask(t: Task) {  
    t.run();  
}  
  
runTask(new Task());  
runTask(new Vehicle()); // Will be a compile time error
```

Argument of type 'Vehicle' is not assignable to parameter of type 'Task'.

Property 's' is missing in type 'Vehicle' but required in type 'Task'.(2345)



References for TypeScript

- <http://www.typescriptlang.org>
- <https://www.tutorialspoint.com/typescript/index.htm>