

Q

**Can JavaScript be made less
rubbish?**

JS strict mode

Errors are generated for

- any assignment to:
 - a non-writable property,
 - a getter-only property,
 - a non-existing property,
 - a non-existing variable,
 - a non-existing object.
- any deletion of variable, function or undeletable property (e.g. `delete Object.prototype;`)
- usage of forbidden words (`eval`, `arguments`, `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, `yield`)
- duplication of parameters (e.g. `f(p,p);`)
- usage of with operator

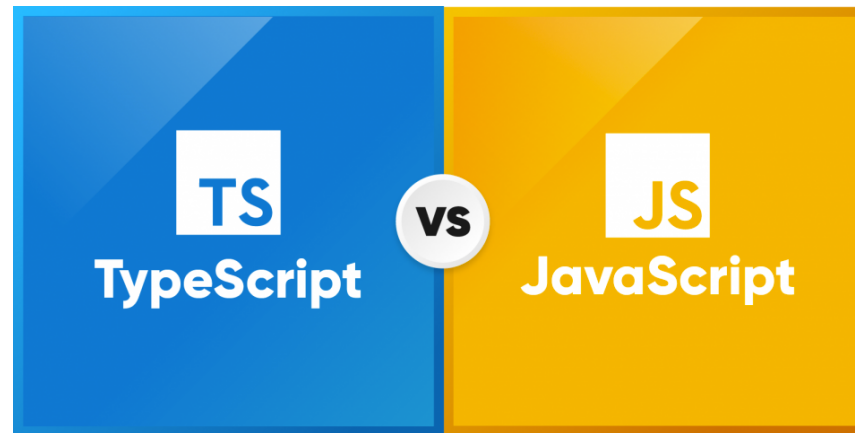
The "use strict" directive is only recognized at the **beginning** of a script or a function.
In a function it has local scope.

see https://www.w3schools.com/js/js_strict.asp

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”.

Making life easier with Typescript



See also <https://www.typescriptlang.org/docs/handbook/>

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 ▾ →


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

EcmaScript 2017

Configuring TypeScript playground

 TypeScript

Download Documentation Handbook Community Playground Tools

Playground TS Config ▾ Examples ▾ What's New ▾

Close

TS Config

Lang TypeScript ▾	Target: ES3 ▾	JSX: None ▾	Module: None ▾
Which language should be used in the editor	Set the supported JavaScript language runtime to transpile to	Control how JSX is emitted	Sets the expected module system for your runtime

Configuring TypeScript playground

TS Config ▾ Examples ▾ What's New ▾

Settings

v3.9.2 ▾ Run Export ▾

→

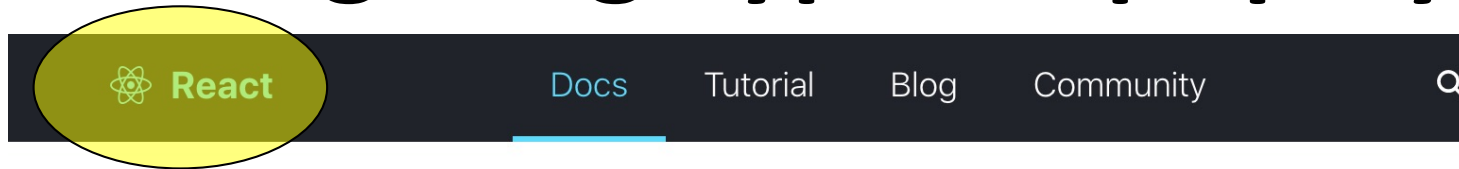
```
1 class Greeting {
2   greet():void {
3     console.log("Hello World!!!")
4   }
5 }
6 var obj = new Greeting();
7 obj.greet();
```

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



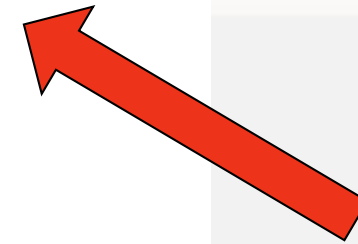
Introducing JSX

Consider this variable declaration:

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

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of 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'];
```

Function typing

We define the type of the variables:

name:type

The screenshot displays the TypeScript Playground interface. The top navigation bar includes the TypeScript logo, links to Download, Docs, Handbook, Community, and Tools, and a search bar labeled 'Search Docs'. Below this, a secondary bar contains 'Playground', 'TS Config', 'Examples', 'What's New', and 'Settings'. The main editor area is split into two panes. The left pane, labeled 'v3.9.2', 'Run', and 'Export', 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(7,18);
5 document.write("result="+result);
6
7 function doSomething():void
8 { document.write("<h1>hola</h1>") }
9
10 doSomething();
11
```

The right pane, labeled 'JS', 'DTS', 'Errors', 'Logs', and 'Plugins', shows the JavaScript output of the code:

```
"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 the TypeScript Playground interface. The left pane 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 right pane shows the JavaScript output under the 'JS' tab:

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

A red error bar is visible on the right side of the code editor, indicating a compilation error.

Result: NaN

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.

Playground

TS Config ▾ Examples ▾ What's New ▾

Settings

v3.9.2 ▾ Run Export ▾ →

```
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
```

JS DTS Errors Logs Plugins

```
"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).

Playground

TS Config ▾ Examples ▾ What's New ▾

Settings

v3.9.2 ▾ Run Export ▾ →

```
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
```

JS DTS Errors Logs Plugins

```
"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));
```



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

The image shows two screenshots of the Visual Studio Code editor interface, illustrating a TypeScript error and its resolution.

Top Screenshot:

- Left Panel (Editor):** Shows a TypeScript file with two lines of code:

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

The variable `myNumber` is declared as `number` but assigned the string value `"pippo"`. A red squiggly line is under the string `"pippo"`.
- Right Panel (Output):** Shows the **JS** tab with the following code:

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

Bottom Screenshot:

- Left Panel (Editor):** The same code is shown. A tooltip is visible over the second line, displaying the error message: `Type '"pippo"' is not assignable to type 'number'. (2322)`. Below the message are the links `Peek Problem` and `No quick fixes available`.
- Right Panel (Output):** The same **JS** tab is shown with the same code as in the top screenshot.

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`.

Access Modifiers in TypeScript

- Public** ← By default all members (properties/fields and methods/functions) of classes are **Public** - accessible internally and externally from outside of the class.
- Private** ← Private members **can not accessible from outside of the class. It can accessible only internally within the class.**
- Protected** ← Protected members **are accessible only internally within the class or any class that extends it but not externally.**

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 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

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

Usages:

explicit form:

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

implicit form:

```
let output = identity("myString"); // ^ = let output: string
```

see <https://www.typescriptlang.org/docs/handbook/generics.html>

Other class related issues

❖ TypeScript doesn't support multiple inheritance.

❖ `super()`

The super call must supply all parameters for base class.
The constructor is not inherited.

❖ 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!

Playground

TS Config ▾ Examples ▾ What's New ▾

Settings

v3.9.2 ▾ Run Export ▾ →

```
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())
```

JS DTS Errors Logs Plugins

```
"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  
}
```

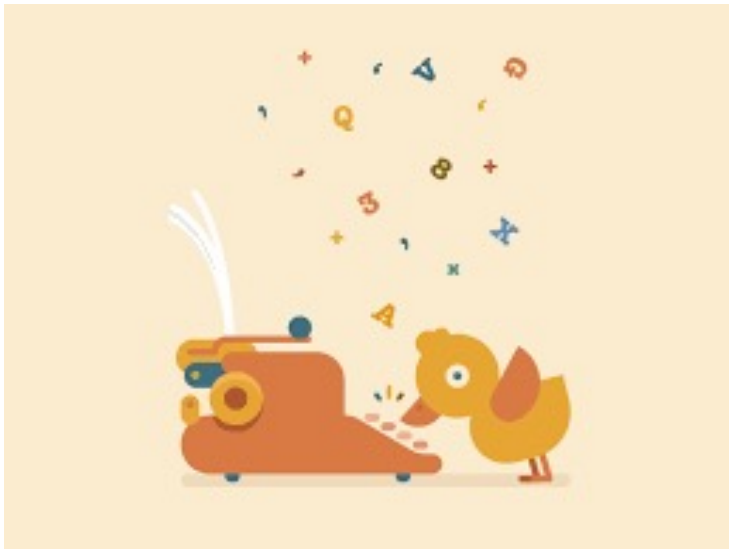
```
interface Child extends IParent1, IParent2 { }  
var Iobj:Child = { v1:12, v2:23}  
console.log("value 1: "+this.v1+" value 2: "+this.v2)
```

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.



Duck-Typing is a **method/rule** used to check the **type compatibility** for more complex variable types. **TypeScript** uses the duck-typing method to compare one object with other objects by checking that both objects have the same type matching names or not.

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>