Learn X in Y minutes

Share this page

Select theme: light dark

Where X=TypeScript

Get the code: learntypescript.ts

TypeScript is a language that aims at easing development of large scale applications written in JavaScript. TypeScript adds common concepts such as classes, modules, interfaces, generics and (optional) static typing to JavaScript. It is a superset of JavaScript: all JavaScript code is valid TypeScript code so it can be added seamlessly to any project. The TypeScript compiler emits JavaScript.

This article will focus only on TypeScript extra syntax, as opposed to JavaScript.

To test TypeScript's compiler, head to the Playground where you will be able to type code, have auto completion and directly see the emitted JavaScript.

```
// There are 3 basic types in TypeScript let isDone: boolean =
false; let lines: number = 42; let name: string = "Anders"; //
But you can omit the type annotation if the variables are
derived // from explicit literals let isDone = false; let lines
= 42; let name = "Anders"; // When it's impossible to know,
there is the "Any" type let notSure: any = 4; notSure = "maybe a
string instead"; notSure = false; // okay, definitely a boolean
// Use const keyword for constants const numLivesForCat = 9;
numLivesForCat = 1; // Error // For collections, there are typed
arrays and generic arrays let list: number[] = [1, 2, 3]; //
Alternatively, using the generic array type let list:
Array<number> = [1, 2, 3]; // For enumerations: enum Color {
Red, Green, Blue }; let c: Color = Color.Green; // Lastly,
"void" is used in the special case of a function returning
nothing function bigHorribleAlert(): void { alert("I'm a little
annoying box!"); } // Functions are first class citizens,
support the lambda "fat arrow" syntax and // use type inference
// The following are equivalent, the same signature will be
inferred by the // compiler, and same JavaScript will be emitted
let f1 = function (i: number): number { return i * i; } //
Return type inferred let f2 = function (i: number) { return i *
i; } // "Fat arrow" syntax let f3 = (i: number): number => {
return i * i; } // "Fat arrow" syntax with return type inferred
let f4 = (i: number) => { return i * i; } // "Fat arrow" syntax
with return type inferred, braceless means no return // keyword
```

```
needed let f5 = (i: number) => i * i; // Interfaces are
structural, anything that has the properties is compliant with
// the interface interface Person { name: string; // Optional
properties, marked with a "?" age?: number; // And of course
functions move(): void; } // Object that implements the "Person"
interface // Can be treated as a Person since it has the name
and move properties let p: Person = { name: "Bobby", move: () =>
{ } }; // Objects that have the optional property: let
validPerson: Person = { name: "Bobby", age: 42, move: () => { }
}; // Is not a person because age is not a number let
invalidPerson: Person = { name: "Bobby", age: true }; //
Interfaces can also describe a function type interface
SearchFunc { (source: string, subString: string): boolean; } //
Only the parameters' types are important, names are not
important. let mySearch: SearchFunc; mySearch = function (src:
string, sub: string) { return src.search(sub) != -1; } //
Classes - members are public by default class Point { //
Properties x: number; // Constructor - the public/private
keywords in this context will generate // the boiler plate code
for the property and the initialization in the // constructor.
// In this example, "y" will be defined just like "x" is, but
with less code // Default values are also supported
constructor(x: number, public y: number = 0) { this.x = x; } //
Functions dist() { return Math.sqrt(this.x * this.x + this.y *
this.y); } // Static members static origin = new Point(0, 0); }
// Classes can be explicitly marked as implementing an
interface. // Any missing properties will then cause an error at
compile-time. class PointPerson implements Person { name: string
move() {} } let p1 = new Point(10, 20); let p2 = new Point(25);
//y will be 0 // Inheritance class Point3D extends Point {
constructor(x: number, y: number, public z: number = 0) {
super(x, y); // Explicit call to the super class constructor is
mandatory } // Overwrite dist() { let d = super.dist(); return
Math.sqrt(d * d + this.z * this.z); } } // Modules, "." can be
used as separator for sub modules module Geometry { export class
Square { constructor(public sideLength: number = 0) { } area() {
return Math.pow(this.sideLength, 2); } } } let s1 = new
Geometry.Square(5); // Local alias for referencing a module
import G = Geometry; let s2 = new G.Square(10); // Generics //
Classes class Tuple<T1, T2> { constructor(public item1: T1,
public item2: T2) { } } // Interfaces interface Pair<T> { item1:
T; item2: T; } // And functions let pairToTuple = function <T>
(p: Pair<T>) { return new Tuple(p.item1, p.item2); }; let tuple
= pairToTuple({ item1: "hello", item2: "world" }); // Including
references to a definition file: /// <reference
path="jquery.d.ts" /> // Template Strings (strings that use
backticks) // String Interpolation with Template Strings let
```

```
name = 'Tyrone'; let greeting = `Hi ${name}, how are you?` //
Multiline Strings with Template Strings let multiline = `This is
an example of a multiline string`; // READONLY: New Feature in
TypeScript 3.1 interface Person { readonly name: string;
readonly age: number; } var p1: Person = { name: "Tyrone", age:
42 }; p1.age = 25; // Error, p1.x is read-only var p2 = { name:
"John", age: 60 }; var p3: Person = p2; // Ok, read-only alias
for p2 p3.age = 35; // Error, p3.age is read-only p2.age = 45;
// Ok, but also changes p3.age because of aliasing class Car {
readonly make: string; readonly model: string; readonly year =
2018; constructor() { this.make = "Unknown Make"; // Assignment
permitted in constructor this.model = "Unknown Model"; //
Assignment permitted in constructor } } let numbers:
Array<number> = [0, 1, 2, 3, 4]; let moreNumbers:
ReadonlyArray<number> = numbers; moreNumbers[5] = 5; // Error,
elements are read-only moreNumbers.push(5); // Error, no push
method (because it mutates array) moreNumbers.length = 3; //
Error, length is read-only numbers = moreNumbers; // Error,
mutating methods are missing
```

Further Reading

- TypeScript Official website
- TypeScript language specifications
- Anders Hejlsberg - Introducing TypeScript on Channel 9
- Source Code on GitHub
- Definitely Typed - repository for type definitions

Got a suggestion? A correction, perhaps? Open an Issue on the Github Repo, or make a pull request yourself!

Originally contributed by Philippe Vlérick, and updated by 12 contributor(s).