TypeScript

# General

TypeScript is an open-source language which builds on JavaScript by adding static type defintions. Each type definition describes the shape of an object, providing better documentation and allowing TypeScript to validate that your code is working correctly.

If valid, TypeScript is compiled in JavaScript by the TypeScript compiler or babel.

# Defining Types

There are several different ways to define the type of an object in TypeScript:

### Type by Inference

Since typescript knows javascript primitive types, if strict mode if off, typescript can infere what type the variable is. For example:

let helloWorld = "Hello World";

// ^ = let helloWorld: string

### Explict Type Definition

Types can be explictly defined on variables or in objects using the typescript TypeName syntax. For example:

let helloWorld: string = "Hello World";

There are two other ways to create complex types, interface and type aliases. Interfaces should be prefered, with type being used for specific features.

## Built in Types

TypeScript has some various built in types:

* boolean - true/false
* bigint - big integers
* number - floating point or bigint (use bigint for bigint)
* string - 'Im a string'
* array - either define as <element-type>[] e.g. number[], or generic Array<type>
* tuple - value sets e.g. [string, number]
* enum - set of numberic values, by default number start at 0, for example:

enum Color {

Red,

Green,

Blue,

}

let c: Color = Color.Green;

* symbol
* object - nonprimitive type
* undefined
* null
* any - allow anything
* unknown - ensure some using this type declares the type
* never - this type cannot happen, commonly used in functions which never end (i.e. throw errors or loop infinitley)
* void - function which has no return value

Note types are case sensitive. Primitive types should only be used in lowercase.

## Interfaces

Interfaces give better error messages than type aliases. For example, creating an interface for a User:

interface User {

name: string;

id: number;

}

interface Admin extends User {

admin: boolean;

}

const user: User = {

name: "Hayes",

id: 0,

};

Now if an object is assigned to the user variable which does not match the User interface an error will be thrown. Interfaces like these can also be applied to JS classes.

When creating a function, parameters and return values will need to be typed. For example:

function getAdminUser(): User {

//...

}

function deleteUser(user: User): void {

// ...

}

### Excess Property Checking

Excess property checking occurs when object literals are assigned to other variables or passed as arguments. This means if the object literal has any additional properties which the target type does not have, an error will be thrown.

If required, avoid this using a type assertion, or by adding a string index signature, which allows any additonal properties:

interface SquareConfig {

color?: string;

width?: number;

[propName: string]: any;

}

### Notes

* Optional properties: properties can be made optional with <name>?:
* Readonly properties: properties can be made readonly using the 'readonly' keyword, prevents mutation methods as well as reassignment

### Functions

To describe a function as a type, a call signature if given containing the parameter list and return type. For example:

interface SearchFunc {

(source: string, subString: string): boolean;

}

Function names do not need to match, only the shape of the input.

### Indexable Types

Indexable objects can have interfaces defined for the type of values at specific indexes in the object. This is a common way to create the shape of an array.

Index signatures can be string or number and will define what is returned for each type of index access. For example, when the following array is indexed by number, it will return a string:

interface StringArray {

[index: number]: string;

}

let myArray: StringArray;

myArray = ["Bob", "Fred"];

### Classes

In typescript javascript classes implement an interface. The interface can then define properties and method in the class. For example:

interface ClockInterface {

currentTime: Date;

setTime(d: Date): void;

}

class Clock implements ClockInterface {

currentTime: Date = new Date();

setTime(d: Date) {

this.currentTime = d;

}

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

}

Interfaces only describe the public interface of a class, not the private side.

In typescript, by default each member of a class is public. Members can be made private using the 'private' keyword. Private memebers are not interchangeable, for two shapes to be compatible they both must have a private member which originated in the same declaration.

Protected memebers can only be accessed from within the class and subclass.

Static properties are memebers on the class, they can be accessed using the '<classname>' keyword instead of the 'this' keyword. For example:

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

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

Classes have two types, the static side and instance side. When a class implements an interface, only the instance side is checked.

## Type ailases

The type keyword is also used to create and extend types:

type User = {

name: string;

id: number;

}

type Admin = { admin: boolean } & User;

The main difference is type aliases are closed and therefore cannot be extended by a second declaration, whereas interfaces can be extended by a second declaration.

## Type Assertions

Type assertions indicate to the compiler that a variables is a specific type, and the compiler will not check it. This is useful if the developer knows more about the type than the compiler.

There are two forms: as keyword, and 'angle-bracket' syntax. For example:

let someValue: unknown = "this is a string";

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

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

## Combining Types

Simple types can be combind using either Unions or Generics.

### Unions

Unions declare that the new type can be one of many other types. For example, setting a list of allowed strings or integers for type:

type WindowStates = "open" | "closed" | "minimized";

type OddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;

Unions can be useful in function declaration where the input can be a singluar or array:

function getLength(obj: string | string[]) {

return obj.length;

}

When unioning interfaces, only members common to all types can be accessed. To access members of specific types in a union , use a switch on a common member which defines a specific state, then access the unique member.

--strictNullChecks can be used to check whether a switch statement is exhastive, since if there is a type not covered, the function will return null instead of the defined type.

### Intersections

Intersection types combine multiple types into one. Therefore, where union types can only use common members, an intersection type will use all member inherited.

Intersections are defined using ampersand:

type ArtworksResponse = ArtworksData & ErrorHandling;

### Generics

Generics provide variables to types, allowing for dynamic interfaces to be created. For example, a common use is to add generics to an Array to define what it can contain:

type StringArray = Array<string>;

type NumberArray = Array<number>;

type ObjectWithNameArray = Array<{ name: string }>;

An example idenitiy function would be:

function identity<T>(arg: T): T {

return arg;

}

Generic classes can also be made. For example, this can also accept strings:

class GenericNumber<T> {

zeroValue: T;

add: (x: T, y: T) => T;

}

let myGenericNumber = new GenericNumber<number>();

myGenericNumber.zeroValue = 0;

myGenericNumber.add = function (x, y) {

return x + y;

};

### Extensions

Extending two interfaces will combine them into a new one. For example:

interface Square extends Shape, PenStroke {

sideLength: number;

}

## Functions

Functions require types on parameters and return value:

function add(x: <parameter-type>): <return-type> {

return x + y;

}

For example:

function add(x: number, y: number): number {

return x + y;

}

let myAdd = function (x: number, y: number): number {

return x + y;

};

In TypeScript every parameter is assumed to be required by the function. Therefore, the number of arguments must match the number of parameters the function expects.

Default parameters come after the required parameters and are optional. Type will be infered if not explicty defined in full function defintion.

### Full function type

A full function type is the shape of a function variable and the function itself, and looks like:

let myAdd: (x: number, y: number) => number = function (

x: number,

y: number

): number {

return x + y;

};

While full function typing is correct, the TypeScript compiler can infer variable type from the function type. Therefore, it is not always required.

### Rest parameters

Rest parameters are the boundless number of optional parameters defined at the end of the parameter list. These can be grouped together into an arrary. For example:

function buildName(firstName: string, ...restOfName: string[]) {

return firstName + " " + restOfName.join(" ");

}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;

### this

'this' can be explictly defined at the beginning of a parameter list. For example:

interface Deck {

suits: string[];

cards: number[];

createCardPicker(this: Deck): () => Card;

}

When using callbacks, 'this' can be set to void to ensure that if the callback required 'this' an error will be thrown.

### Overloads

Overloads are used to supply multiple function types to the same function, allowing for different input shapes to return different output shapes. For example:

function pickCard(x: { suit: string; card: number }[]): number;

function pickCard(x: number): { suit: string; card: number };

function pickCard(x: any): any {

// Check to see if we're working with an object/array

// if so, they gave us the deck and we'll pick the card

if (typeof x == "object") {

let pickedCard = Math.floor(Math.random() \* x.length);

return pickedCard;

}

// Otherwise just let them pick the card

else if (typeof x == "number") {

let pickedSuit = Math.floor(x / 13);

return { suit: suits[pickedSuit], card: x % 13 };

}

}

## Structural Type System

One of typescripts core principles is that type checking focuses on the shape values have. This means if two objects have the same shape they are considered to be of the same type. Therefore, if not in strict mode and a variable is defined with the same shape as a type, it will be accepted as that type. For example, the following will not thrown an error:

interface Point {

x: number;

y: number;

}

function logPoint(p: Point) {

console.log(`${p.x}, ${p.y}`);

}

// logs "12, 26"

const point = { x: 12, y: 26 };

logPoint(point);

With the way this interface is defined, the type checking only requires a subset of the objects properties to match, so the following will also be a valid Point:

const point3 = { x: 12, y: 26, z: 89 };

logPoint(point3); // logs "12, 26"

const rect = { x: 33, y: 3, width: 30, height: 80 };

logPoint(rect); // logs "33, 3"