The most basic types in Typescript as basic primitives in Javascript:

* bigint: 0n, 2n, -4n, …
* boolean: true or false
* null
* number:0 , 2, -4 …
* string: “helloworld”
* symbol: Symbol(), Symbol("hi"), …
* undefined

example:

1337n; // bigint true; // boolean null; // null 1337; // number "Louise"; // string Symbol("Franklin"); // Symbol undefined; // undefined

At its core, TypeScript’s type system works by:

1. Reading in your code and understanding all the types and values in existence
2. For each object, seeing what type its initial declaration indicates it may contain
3. For each object, seeing all ways it’s used later on
4. Complaining to the user if an object’s usage doesn’t match with its type

Take the following snippet, in which TypeScript is emitting a type error:

let firstName = "Whitney";

firstName.length();

// This expression is not callable.

// Type 'Number' has no call signatures

**Kinds of Errors** While writing TypeScript, the two kinds of “errors” you’ll come across most frequently are:

* Syntax: blocking TypeScript from being converted to JavaScript.
* Type: something mismatched has been detected by the type checker.

Syntax Errors:

Syntax errors are when TypeScript detects incorrect syntax that it cannot understand as code.

let let wat;

// Error: ',' expected.

Type Errors

Type errors occur when your syntax is valid but the TypeScript type checker has detected an error with the program’s types.

console.blub("Nothing is worth more than laughter.");

// Error: Property 'blub' does not exist on type 'Console'.

Assignability

TypeScript’s checking of whether a value is allowed to be provided to a function call or variable is called “assignability”: whether that value is assignable to the location it’s passed to

let car:string = "mercedes";

//car = 44;

// Error: Type 'number' is not assignable to type 'string'

Type Annotations

It’ll consider them to be implicitly the any type: a type indicating that it could be anything in the world.

Instead, TypeScript provides a syntax for declaring the type of a variable, using what’s called a type annotation.

let rocker: string;

rocker = "Joan Jett";

Types Shapes

TypeScript doesn’t only check that the values assigned to variables match their original types: it also knows what member properties should exist on objects

let cher = {

firstName: "Cherilyn",

lastName: "Sarkisian",

};

cher.middleName;

// Property 'middleName' does not exist on type

// '{ firstName: string; lastName: string; }'.

Chapter 3. Unions and Narrowing

**Unions**: Expanding a value’s allowed type to be two or more possible types.

Take this mathematician variable:

let mathematician : string | number  = Math.random() > 0.5 ? 9001 : "Mark Goldberg";

console.log(mathematician)

let thinker: boolean | string = false;

if (Math.random() > 0.5) {

 thinker = "Susanne Langer";

}

**Narrowing**: Reducing a value’s allowed type to not be one or more possible types.

Assignment narrowing comes into play when a variable is given an explicit

union type annotation and an initial value, too. TypeScript will understand

that while the variable may later receive a value of any of the union typed

values, it starts off as only the type of its initial value.

In the following snippet, inventor is declared as type number |

string, but TypeScript knows it’s immediately narrowed to a string

from its initial value:

let inventor: number | string = "Hedy Lamarr";

inventor.toUpperCase(); // Ok: string

inventor.toFixed();

// ~~~~~~~

// Error: Property 'toFixed' does not exist on type 'string'

let scientist : string ;

scientist = "string"

let text  = scientist;

if (typeof text === "string") {

  text.toUpperCase(); // Ok: string

} else {

  text.toFixed(); // error

}

function padLeft(padding: number | string, input: string) {

  return " ".repeat(padding) + input;

}

 /\*Argument of type 'string | number' is not assignable to parameter of type 'number'.

  Type 'string' is not assignable to type 'number'. \*/

Chapter 4. Literals

literal types: more specific versions of primitive types.

const car : null = null;

const jet : undefined = undefined;

const angka : number = 7;

const trueOrfalse:boolean | string = true && "4";

const mySymbol = Symbol()

var bigBin:bigint = BigInt("0b1010101001010101001111111111111111");

const array = [4,"string" , true]

const check =mySymbol;

console.log(typeof array)

console.log(bigBin);

Primitive:- (String,Boolean,Number,BigInt,Null,Undefined,Symbol )

Non-Primitive:- Object (array, functions) also called object references.

//kind type of

// number,boolean,string, object , undefined , symbol , bigint

Chapter 5. Functions

TypeScript allows you to declare the type of function parameters with a type annotation.

In theory, you don’t need to add proper type annotations to function parameters for your code to be valid TypeScript syntax. TypeScript might yell at you with type errors but the emitted JavaScript will still run.

function singAllTheSongs(singer: string, ...songs: string[]):void {

console.log(singer) //"Alicia Keys"

console.log(songs) // ["Bad Romance", "Just Dance", "PokerFace"]

}

singAllTheSongs("Alicia Keys");

singAllTheSongs("Lady Gaga", "Bad Romance", "Just Dance", "PokerFace");

let createStrings: () => string[];

let insecure: () => string[] | number[];

insecure = () => {

    return ["string" ]

}

Parameter **Type Inferences**

It would be cumbersome if we had to declare parameter types for every function we write

Functions set as values for previously declared variables don’t need to have their types declared:

const songs = ["Call Me", "Jolene", "The Chain"];

let singer: (song: string[]) => void;

singer = function (song) {

 console.log(`${song}`);

};

singer(songs);

Functions immediately passed to parameters have function parameter types inferred as well:

let text = "";

const fruits = ["apple", "orange", "cherry"];

//array.forEach(function(currentValue, index))

let myFunction : (items:string , index:number) => void;

myFunction = (items,index) => {

  text += " " + items;

}

fruits.forEach(myFunction);

console.log(text);

Void Returns

They either have no return statements or only have return statements that don’t return a value. TypeScript allows using a void keyword to refer to the return type of a function that returns nothing.

function logSong(song: string | undefined | boolean): void {

 if (!song) {

 return; // Ok

 }

 console.log(`${song};`)

 return true; // Error: Type 'boolean' is not assignable to type 'void'.

}

void is not the same as undefined. void means the return type of a function will be ignored, while undefined is a literal value to be returned will be Error.

Functions don’t need to actually return void in order to be used in locations declared to be a function type with a void. Remember, it’s an indication that a function’s returned value isn’t meant to be used.

Chapter 6. Arrays

let arrayOfNumbers: number[];

arrayOfNumbers = [4, 8, 15, 16, 23, 42];

If you don’t include that type annotation on a variable initially set an empty array, TypeScript will treat the array as implicitly any[], meaning it can receive any content.

Multi-Dimensional Arrays

A 2D array, or an array of arrays, will have two `[]`s:

let arrayOfArraysOfNumbers: number[][];

arrayOfArraysOfNumbers = [

 [1, 2, 3],

 [2, 4, 6],

 [3, 6, 9],

];

A 3D array, or an array of arrays of arrays, will have three `[]`s. 4D arrays have four `[]`s. 5D arrays have five `[]`s. You can guess where this is going for 6D arrays and more.

Union Type Arrays

JavaScript arrays can contain all sorts of different types of values, as you saw in the beginning of this chapter. You can use a union type to indicate that each member of an array is either one type or another.

When using array types with unions, parenthesis () may be used to indicate which part of an annotation is the contents of the array or the surrounding union type:

// Type is either a number or an array of strings

let numberOrStrings: number | string[];

// Type is an array of items that are each either a number or a string

let stringCreators: (number | string)[];

// Type is (string | undefined)[]

const namesMaybe = [

 "Aqualtune",

 "Blenda",

 undefined,

];

// Type is (string | date)[]

const soldiersOrDates: (string | Date)[] = ["Deborah Sampson", new Date(1782, 6,3)];

Array Members

TypeScript understands typical index-based access for retrieving members of an array to give back an item of that an array’s type.

Unsound Members

function withItems(items: string[]) {

 console.log(items[9001].length);

}

withItems([]); // cannot read properties of undefined (reading ‘length’)

We can see that it’ll crash at runtime with `Cannot read property length of

undefined`, but TypeScript intentionally does not make sure retrieving array

members exist

Array members are an example of an area where the TypeScript type system

is known to be technically unsound: it can get types mostly right, but

sometimes it messes up

Spreads and Rests

If two arrays of different types are spread together to create a new array, the new array will be understood to be a union type array of elements that are either of the two original types:

// Type is (number | string)[]

const soldiers = ["Harriet Tubman", "Joan of Arc", "Lozen"];

const soldierAges = [90, 19, 49];

const conjoined = [...soldiers, ...soldierAges];

console.log(conjoined) //["Harriet Tubman", "Joan of Arc", "Lozen", 90, 19, 49]

Rests

One nice part of the design of JavaScript function rest parameters is that they work seamlessly with array values in code. An array may be passed directly as the rest parameter for a function that accepts one, and TypeScript’s type checker will make sure those types match up

function announceNames(...names: string[]) {

 console.log(names)

}

announceNames("warriors"); //[“warriors”]

Tuples

Tuple arrays often have a specific known type at each index which may be more specific than a union type of all possible members of the array.

TypeScript allows describing tuple array types with [, a comma-separated list of the type of each member in the array, and ].

let yearAndWarrior: [number, string];

yearAndWarrior = [530, "Tomyris"]; // Ok

yearAndWarrior = [false, "Tomyris"];

// ~~~~~

// Error: Type 'boolean' is not assignable to type 'number

const three: [boolean, number, string] = [false, 1663,

"Nzingha"];

const two: [boolean, number] = temp1;

Tuples as Rest Parameters

Because tuples have strong typing information about the member at each position of an array, they can be passed as ... rest parameters to functions This can be useful if you’d like to store arguments for a function in arrays and pass them

function logPair(name: string, ...value: number[]) {

 console.log(`${name} has ${value}`);

}

var pairArray : [string,number] = ["Amage", 1];

logPair(...pairArray);

function singAllTheSongs(singer: string, ...songs: string[]):void {

console.log(singer) //"Alicia Keys"

console.log(songs) // ["Bad Romance", "Just Dance", "PokerFace"]

}

//singAllTheSongs("Alicia Keys");

var pairArray : [string , string,string,string] = ["Lady Gaga", "Bad Romance", "Just Dance", "PokerFace"];

//singAllTheSongs("Lady Gaga", "Bad Romance", "Just Dance", "PokerFace");

singAllTheSongs(...pairArray)

function logPair(name: string|number, ...value: (number | string)[]) {

 console.log(`${name} has ${value}`);

}

const pairArray:[string,number] = ["Amage", 1];

logPair(...pairArray);

// Error: A spread argument must either have a tuple type or be passed to a rest parameter.

const pairTupleIncorrect: [number, string] = [1, "Amage"];

logPair(...pairTupleIncorrect);

// Error: Argument of type 'number' is not assignable to parameter of type 'string'.

Tuple Inferences

TypeScript generally treats created arrays as variable length arrays, not tuples.

The following firstCharAndSize function is inferred as returning (number | string)[], not [string, number], because that’s the type inferred for its returned array literal:

// Return type: (number | string)[]

function firstCharAndSize(input: string) : (string | number)[] {

 return [input[0], input.length , input];

}

// firstChar type: number | string ☹

// size type: number | string ☹

const [firstChar, size , omni]: (string|number)[] = firstCharAndSize("Cathay Williams");

const [InitialName, level , DotaName] = firstCharAndSize("Juggernaut");

console.log(firstChar) // "C"

console.log(size) // 15

console.log(omni)

There are two common ways in TypeScript to indicate that a value should be a more specific tuple type instead of a general array type: explicit tuple types and const assertions.

A const assertion is a special type assertion that uses the const keyword instead of a specific type name.

Explicit Tuple Types

Tuple types may be used in type annotations, such as the return type annotation for a function. If the function is declared as returning a tuple type and returns an array literal, that array literal will be inferred to be a tuple instead a more general variable length array

Const Asserted Tuples

Typing out tuple types in explicit type annotations can be a pain for the same reasons as typing out any explicit type annotations. It’s extra syntax for you to write and update as code changes.

As an alternative, TypeScript provides an as const operator that can be placed after a value

If placed after an array literal it will indicate that the array should be treated as a tuple.

// Type: (number | string)[]

const temp1 = [1157, "Tomoe"];

// Type: readonly [1157, "Tomoe"]

const temp2 = [1157, "Tomoe"] as const

In this example, pairMutable is allowed to be modified because it has a traditional explicit tuple type:

const pairMutable: [number, string] = [1157, "Tomoe"];

pairMutable[0] = 1247; // Ok

However, as const`s make the value not assignable to the mutable :

const pairAlsoMutable: [number, string] = [1157, "Tomoe"] as const;

// Error: The type 'readonly [1157, "Tomoe"]' is 'readonly'

// and cannot be assigned to the mutable type '[number, string]'.

conclusion: pairAlsoMutable, and members of the constant pairConst not allowed to be modified

const pairConst = [1157, "Tomoe"] as const;

pairConst[0] = 1157;

// Error: Cannot assign to '0' because it is a read-only property.

In practice, readonly tuples are convenient for function returns. Returned values from functions that return a tuple are often destructured immediately anyway, so the tuple being readonly does not get in the way of using the function.

This firstCharAndSizeAsConst returns a readonly [string, number], but the consuming code only cares about retrieving the values from that tuple:

// Return type: readonly [string, number]

function firstCharAndSizeAsConst(input: string) {

 return [input[0], input.length] as const;

}

// firstChar type: string

// size type: number

//create destructuring variable

const [firstChar, size] = firstCharAndSizeAsConst("John Cena");

Chapter 7. Objects and Interfaces

Interfaces

The most common way to describe an object’s shape in TypeScript is with an interface. An interface is a named list of properties that are expected to exist on an object

The following Poet interface describes any object that has those same born and name member properties:

interface PoetS {

 born: number;

 name: string;

}

function greetPoet(poet:PoetS) {

 // Ok: poet.born is known to be of type `number`

 const born: number = poet.born;

 // Ok: poet.name is known to exist (it's a `string`)

 console.log(`${poet.name}, born in ${born}`);

 console.log(PoetS) //'Poet' only refers to a type, but is being used as a value here.

}

// Ok: the object provided to greetPoet has a born number and a name string

greetPoet({

 born: 1932,

 name: "Sylvia Plath"

})

Structural Typing

interface FirstName {

    firstName:string;

}

interface LastName {

    lastName:string;

}

const hasBoth = {

    firstName: "lucie",

    lastName: "Alina"

}

const takeMyNames = (withFirstName: FirstName) => {

    console.log(withFirstName.firstName);

}

takeMyNames(hasBoth);

TypeScript’s type system is structurally typed: meaning any object that happens to satisfy an interface is allowed to be used as an instance of that interface. In other words, when you declare that a parameter or variable is of a particular interface type, you’re telling TypeScript that whatever object(s) you use there need to have those properties.

Usage Checking

interface StartDate {

 start: Date;

}

const hasStartDate: StartDate = {

 start: "0000-00-00",

 // Error: Type 'string' is not assignable to type 'Date'.

}

interface Poet {

    born:number;

    jancok:string;

}

const myPoet: Poet = {

 born: 1928,

 name: "Maya Angelou"

};

//Type '{ born: number; name: string; }' is not assignable to type 'Poet'. Object literal may only specify known properties, and 'name' does not exist in type 'Poet'

Banning excess properties is another way TypeScript helps make sure your code is clean and does what you expect. Excess properties not declared in their interface types are often unused code that you the author didn’t mean to include.

Types of Properties have 2:

* Optional Properties

As with function parameters, you can include a ? before the : type in an interface property’s type annotation to indicate that it’s an optional property.

interface Placeholder {

  required: boolean;

  optional?: number;

}

const missingRequired: Placeholder = {

    required: true

};

* Readonly Properties

sometimes wish to block users of your interface from reassigning members of that interface. TypeScript allows you to add a readonly modifier before a member name to indicate that once set, that member should not be set to a different value. . These readonly members can be read from normally, but not set to anything new. Readonly have character Immutable

interface Massenger {

    readonly announcement: string;

}

function proclaiz(message: Massenger) {

    //Ok : reading the message propert doesn't attemp to modify

    console.log(message.announcement)

    messenger.announcement += "?";

    // Error: Cannot assign to 'announcement'

   // because it is a read-only property.

}

const messengerIsh = {

 announcement: "Hello, world!",

}

// Ok: messengerIsh is an inferred object type with announcement, not a Messenger

messengerIsh.announcement += "!";

// Ok: proclaim takes in Messenger, which happens to

// be a more specific version of messengerIsh's type

proclaim(messengerIsh);

Readonly interface members are a good way to make sure areas of code

don’t unexpectedly modify objects they’re not meant to

Chapter 8. Classes