TYPESCRIPT

ADVANCED

THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

Kindly go through the following book before reading this.book

INTRODUCTION TO TYPESCRIPT

TYPESCRIPT

TypeScript is described as a “superset of JavaScript” or “JavaScript with types.” It is four things:

>> Programming Language

A language that includes all the JavaScript syntax, plus new TypeScript syntax for types.

>> Type Checker

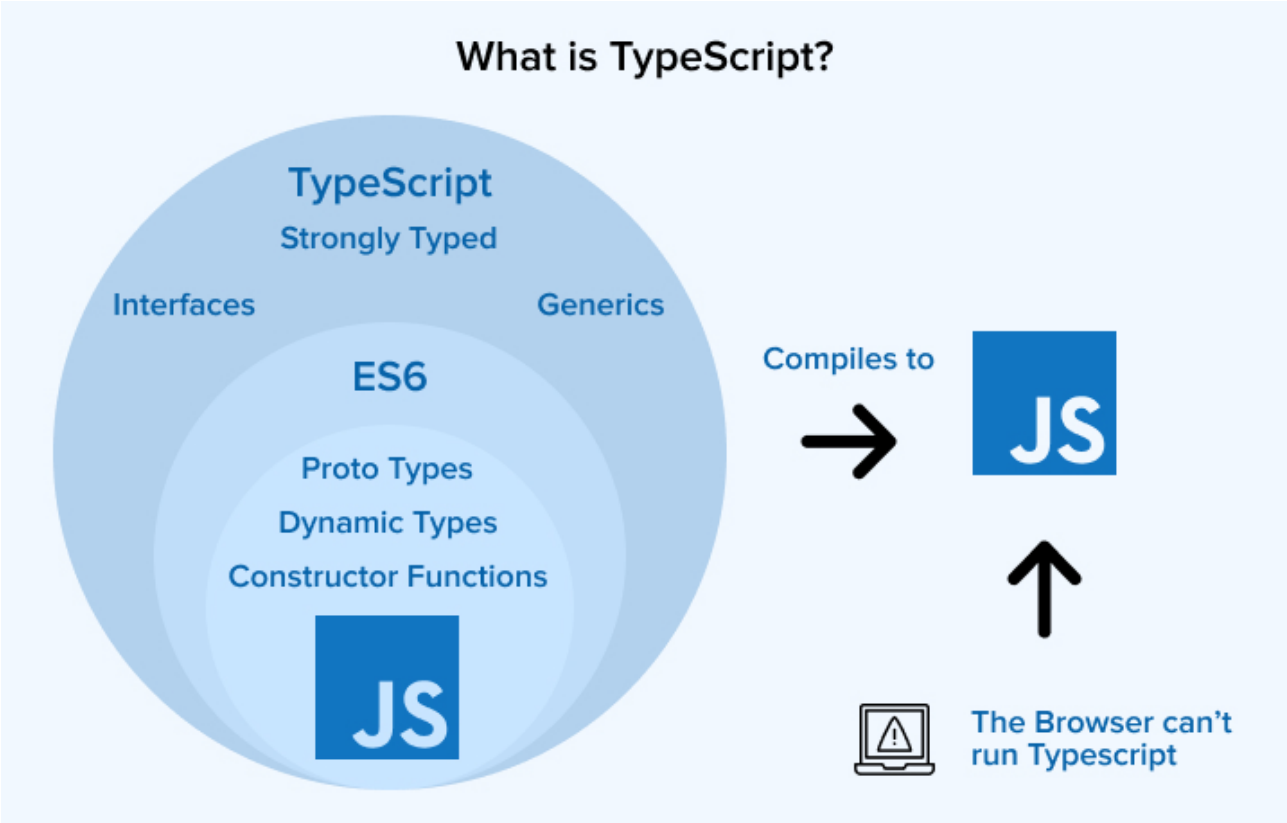
A program that takes in a set of files written in JavaScript and/or TypeScript, develops an understanding of all the constructs created, and lets the developer know if it thinks anything is set up incorrectly.

>> Transpiler

A program that runs the type checker, reports any issues, then outputs the equivalent JavaScript code.

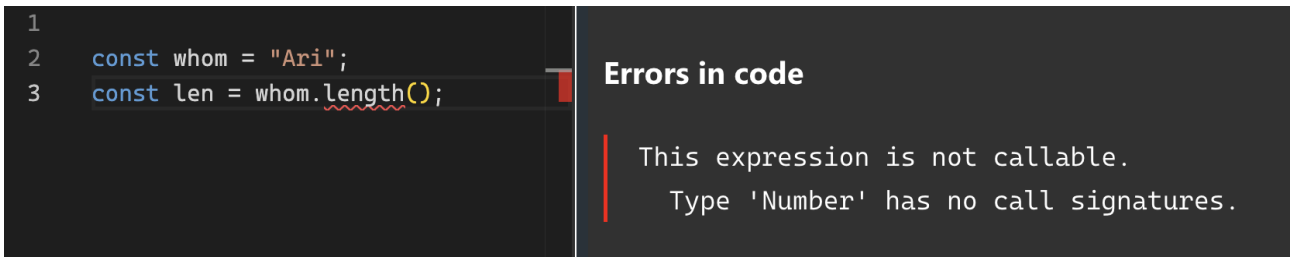
>> Language service

A program that uses the type checker to tell editors how to provide helpful utilities to developers.



EXAMPLES

When we code the following in TypeScript (I am using TypeScript playground),

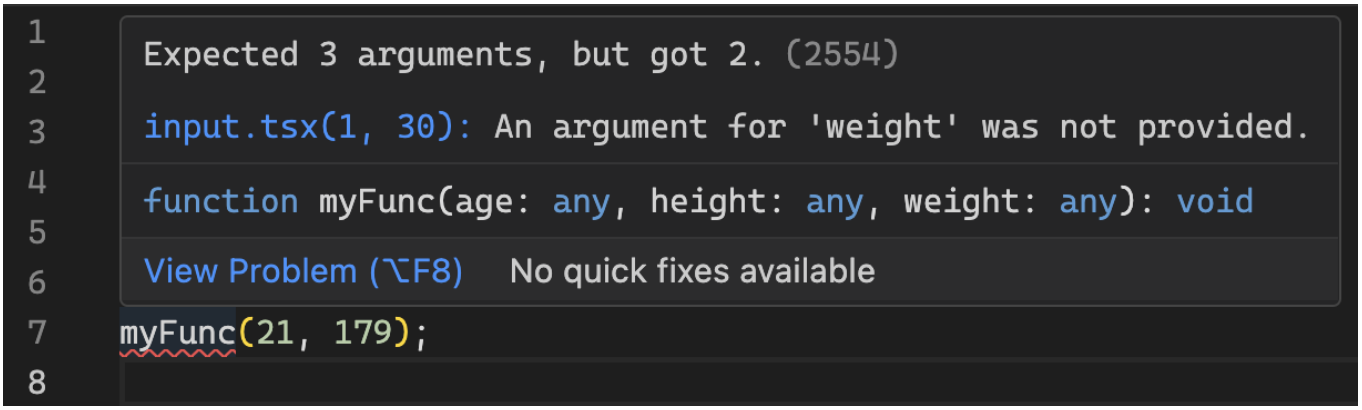


It (The Language Service) will come to us with a complaint. It would use its knowledge that the length property of a string is a number; not a function. We can also hover on the highlight and know what it's all about. TypeScript can give us confidence that changes in one area of code won't break other areas of code that use it.

For an instance, if we change the number of required parameters for a function, TypeScript will let us know if we forget to update a place that calls the function.

```
1   function myFunc(age, height, weight){
2       console.log(age);
3       console.log(height);
4       console.log(weight);
5   }
6
7   myFunc(21, 179);
8
```

We can hover on it and know what happens...



ONCE INFERRED! INFERRED!

Once the type of the data is inferred, it can't be changed later.

```
1
2          let nameOfHim: string
3    let nameOfHim = "Ari";
4
```

It's not possible to give a data of another type later.

```
1
2
3   Type 'number' is not assignable to type 'string'. (2322)
4   let nameOfHim: string
5
6   View Problem (⌥F8)    No quick fixes available
7   nameOfHim = 21;
```

If we didn't specify the data, it will be of any type.

```
3
4          var ageOfHim: any
5    var ageOfHim;
6    let nameOfHim;
```

This can be called, Ever Evolving Variable. Because, it can get data of different type over the time.

```
5     var ageOfHim;
6     let nameOfHim;
7
8     ageOfHim  = 21;
9     ageOfHim = "Ari";
10    ageOfHim = true;
```

It is important to note where we don't have to use the type hints. I mean, reduntantly...

```
1    let firstName: string = "Tilapia";
```

In the statement above, it is redundant to use the type hint.

MODULES

Module is a file with a top-level export or import. Any file that is not a module is a Script. A variable declared in one module with the same name as a variable declared in another file won't be considered a naming conflict as long as one file imports other files variable.

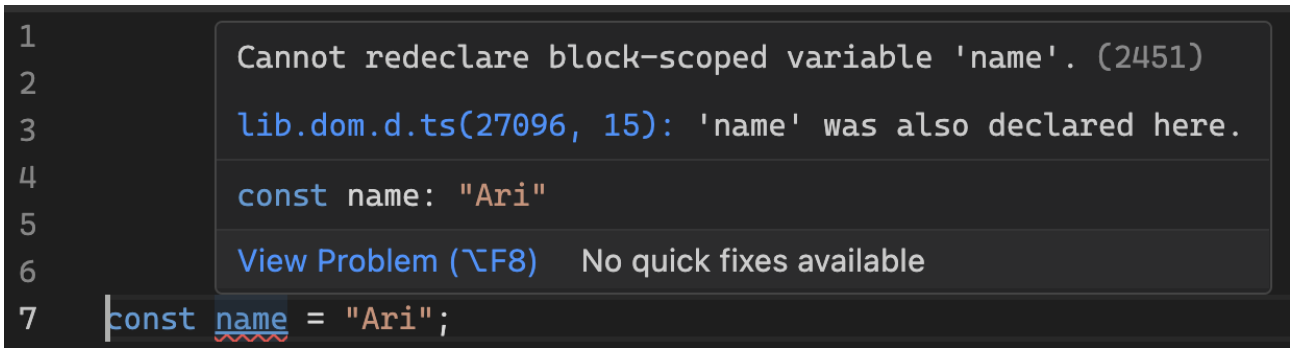
TYPESCRIPT

```typescript
// a.ts
export const shared = "Cher";
// b.ts
export const shared = "Cher";

// c.ts
import { shared } from "./a";
// Error: Import declaration conflicts with local declaration of 'shared'.
export const shared = "Cher";
// Error: Individual declarations in merged declaration
// 'shared' must be all exported or all local.
```

If there is no export or import statements in a file, it is a script.

Script is where, the variables declared in one are visible to others. If the variable name is already declared in another script, we can't declare it in current script.



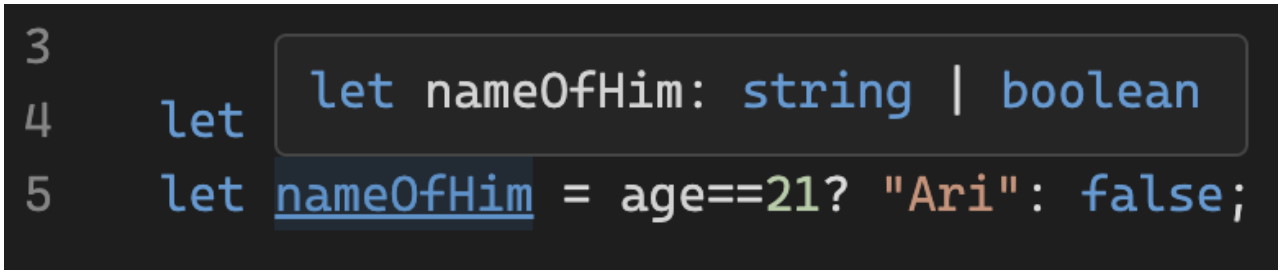
If we want to avoid these kind of stuffs, let's force our script to be a module by having an empty export.

UNIONS

There maybe some cases like the following.

```
4    let age = 21;
5    let nameOfHim = age==21? "Ari": false;
```

What could be the type of nameOfHim when we hover on it? What could be the type inferred? It's Union!



This is an example situation where we can use Union Type.

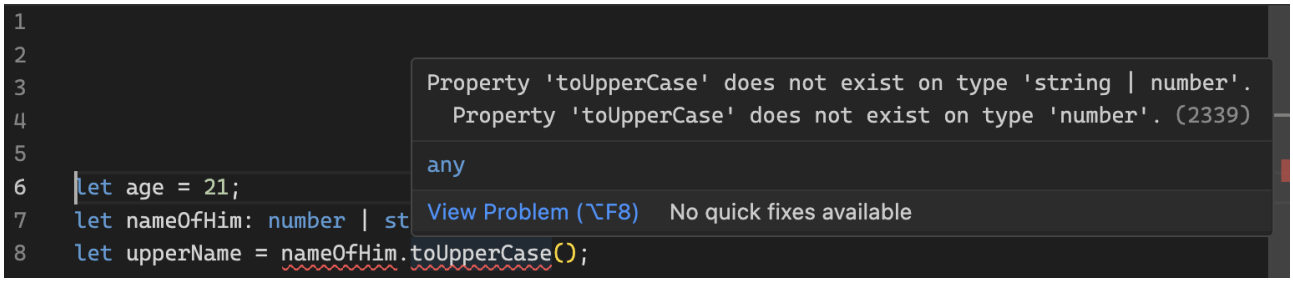
```
2    let isAriEligible: boolean | string = "YES";
3    isAriEligible = true;
```

Yes.. We use the vertical bar (|) to denote the union type.

If we have defined a variable with Union type, the properties available for that variable are the common properties of both the types.

```
1    let age = 21;
2    let nameOfHim: number | string;
3    let upperName = nameOfHim.toUpperCase();
4
```

Now, if we hover on it, it will show like the following.



There will be no errors for properties which are common to both the types.

```
6    let age = 21;
7    let nameOfHim: number | string = "Ari";
8    let upperName = nameOfHim.toString();
```

NARROWING

Narrowing is when TypeScript infers from our code that a value is of a more specific type than what it was defined, declared, or previously inferred as.

```
let nameOfHim: number | string;
nameOfHim = "Ari";
nameOfHim.toUpperCase();
nameOfHim.toFixed();
nameOfHim = 21.23;
nameOfHim.toFixed();
nameOfHim.toUpperCase();
```

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 assigned.

TypeScript is smart enough to understand if we use conditional constructs while invoking properties.

```
1
2   let nameOfHim: number | string;
3   nameOfHim = Math.random()>0.5 ? "ari": 21;
4
5   if(nameOfHim === 21){
6       nameOfHim.toFixed();
7   } else if(nameOfHim == "ari"){
8       nameOfHim.toUpperCase();
9   }
10  nameOfHim.toFixed();
11  nameOfHim.toUpperCase();
12
```

We can also do typeof checks.

```
5   if(typeof nameOfHim === "number"){
6       nameOfHim.toFixed();
7   } else if(typeof nameOfHim == "string"){
8       nameOfHim.toUpperCase();
9   }
```

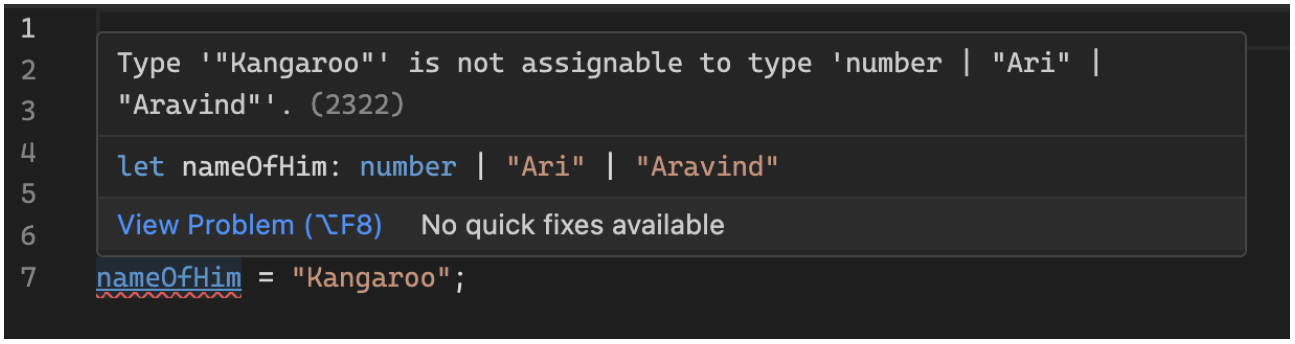
Why not... We can also go with truthiness checks i.e., if(name) {...}

LITERAL TYPES

When we define a variable with const, it's type will be inferred as literal.

```
3   let nameOfHim: number | "Ari" | "Aravind";
4   nameOfHim = "Ari";
5   nameOfHim = 21;
6   nameOfHim = "Aravind";
7   nameOfHim = "Kangaroo";
```

If we hover on the variable,



Literal is a type itself if used in the place where type has to be used.

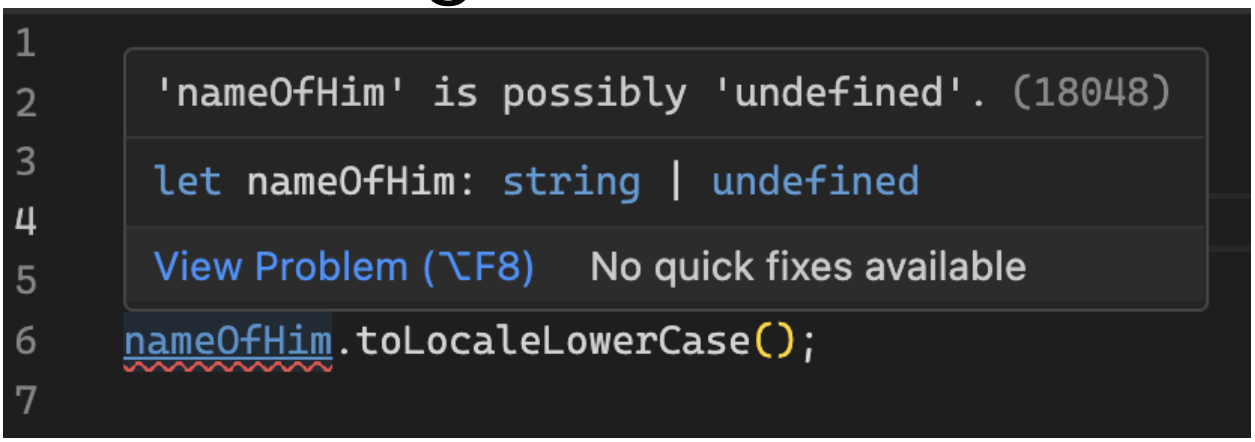
```
1
2   let nameOfHim: "Ari";
3   nameOfHim = "Ari";
4   nameOfHim = "Aravind";
5
```

STRICT NULL CHECKING

In TS, we can enable or disable type checking. Just look at the following code where strict null checking is enabled.

```
1
2    let nameOfHim : string | undefined;
3    nameOfHim.toLocaleLowerCase();
4
```

On hovering,



We can also disable it if needed using the configuring options.

TYPE ALIASES

If we have to use longer union types that are inconvenient to type out repeatedly, we can use type aliases. Actually, it's not only for Unions. We can use it to alias any types.

```
let nameOfHim: string | "ari" | 21 | number | boolean | undefined;
```

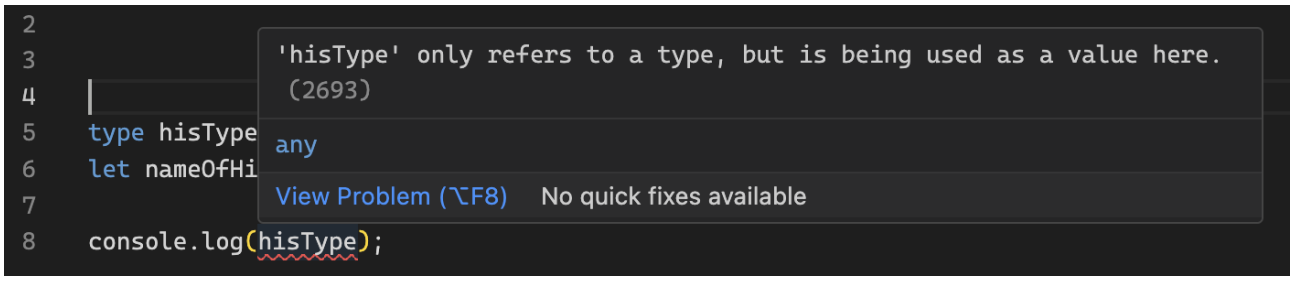
We can shorten this by using a type alias as shown below.

```
type hisType = string | "ari" | 21 | number | boolean | undefined;
let nameOfHim: hisType;
```

Type aliases are a handy feature to use in TypeScript whenever our types start getting complex. For now, that just includes long union types; later on it will include array, function, and object types.

TYPE ALIASES AREN'T JS

Type aliasing feature isn't that of JavaScript. It exists in the type system of TypeScript. Look at the following.



So, what about something like Hoisting? Well! There is no such Hoisting in TypeScript.

When TypeScript compiles our code, it performs a full pass over the entire file, gathering all type information first, including type aliases, interfaces, and class types.

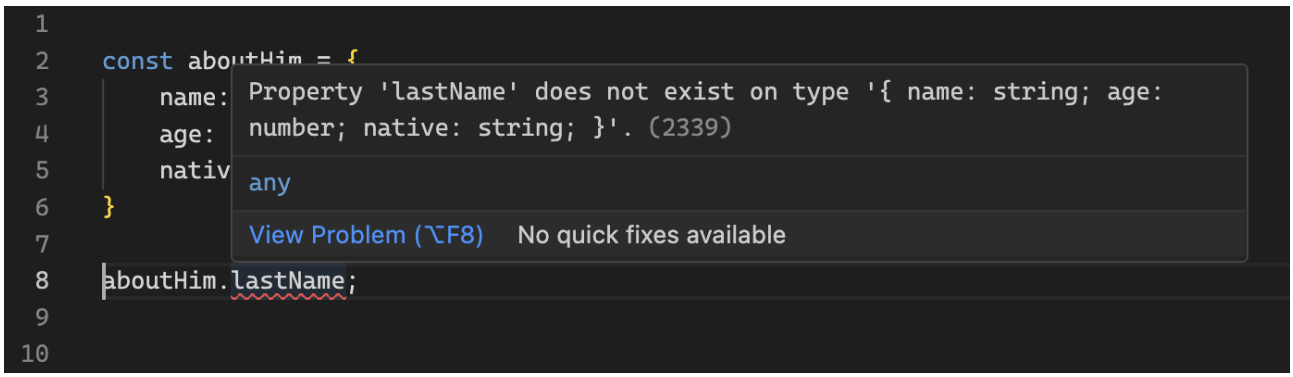
This means that even if a type is declared later in the file, the compiler already "knows" about it during this first pass and can refer to it earlier in the code.

```
4    type IdMaybe = Id | undefined | null;
5    type Id = number | string;
```

OBJECT TYPES

We do start with the example shown above. If it is JavaScript, imagine the situation. We can access aboutHim.lastName, but since the lastName property doesn't exist, it returns undefined without throwing an error. (JS lacks static type checking!) TypeScript enforces strict type checking.

Since aboutHim doesn't have a lastName property, trying to access aboutHim.lastName will cause a compile-time error like:

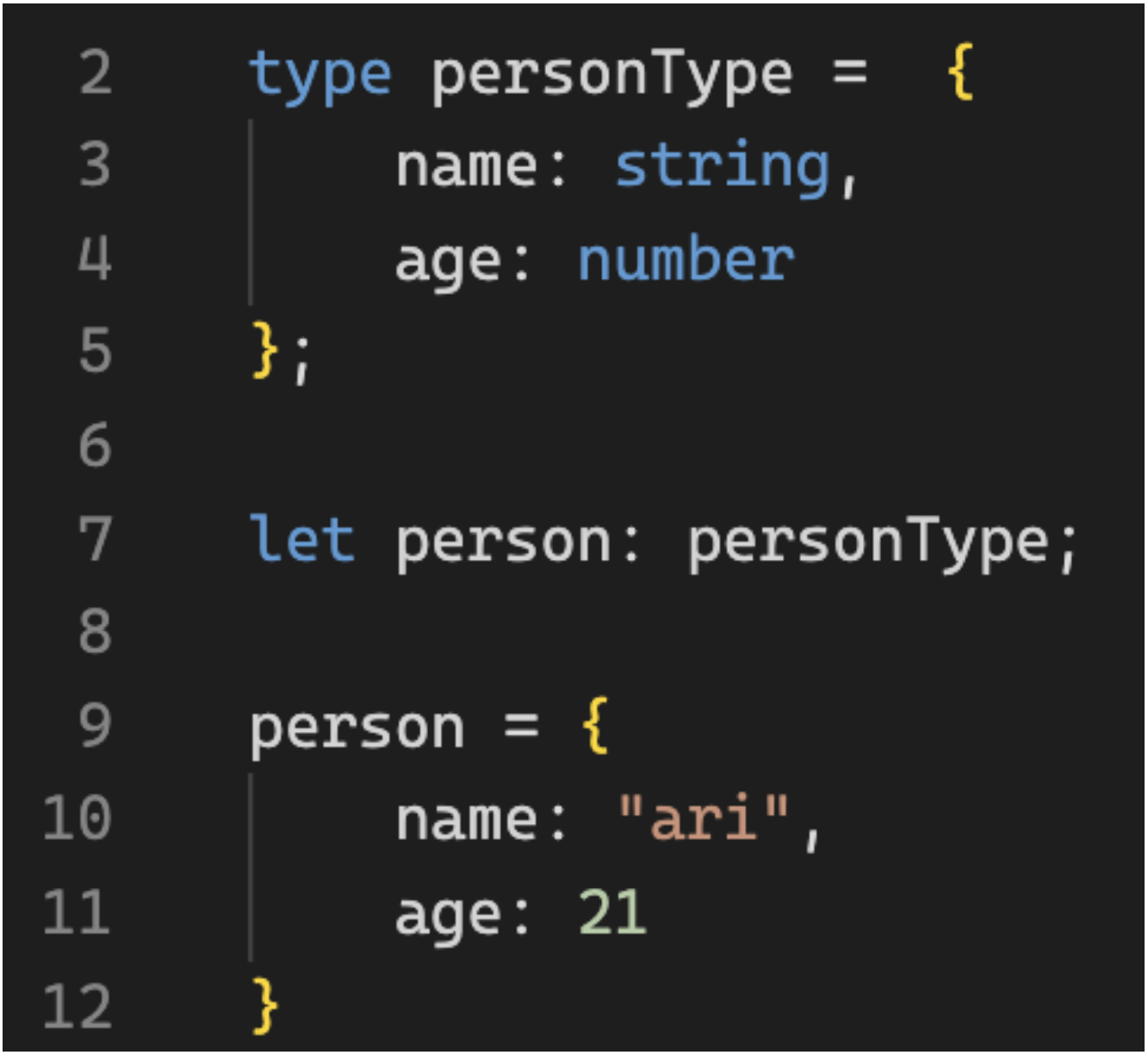


We can declare the types of Object's Properties as shown below.

```
let person: {
    name: string,
    age: number
};

person = {
    name: "ari",
    age: 21
}
```

This is one of the places where we can use type aliasing.



STRUCTURAL TYPING

Structural typing means that two objects are considered compatible as long as they have the same structure, regardless of what they are explicitly typed as. TypeScript checks whether an object has the required properties for a given type, instead of strictly checking if it's an instance of that type.

```
type WithFirstName = { firstName: string };
type WithLastName = { lastName: string };
const hasBoth = { firstName: "ARI", lastName: "ARAN" };
// Ok: `hasBoth` contains a `firstName` property of type `string`
let withFirstName: WithFirstName = hasBoth;
// Ok: `hasBoth` contains a `lastName` property of type `string`
let withLastName: WithLastName = hasBoth;
```

type WithFirstName and type WithLastName define two types. WithFirstName expects an

object with a firstName property of type string. WithLastName expects an object with a lastName property of type string. hasBoth is an object that contains both firstName and lastName properties. When assigning hasBoth to withFirstName, TS checks if hasBoth contains a firstName property of type string. Since it does, the assignment is valid. Similarly, when assigning hasBoth to withLastName, TS checks if hasBoth contains a lastName property of type string. Again, the assignment is valid.

TypeScript doesn't care that hasBoth has extra properties (lastName in the case of withFirstName or firstName in the case of withLastName). As long as the required structure (properties and types) matches, the object can be assigned. This is structural typing at work: it is the shape of the object that matters, not its explicit type.

Types must match with the data! No compromise here!

```
2   type FirstAndLastNames = { first: string; last: string };
3   const hasBoth: FirstAndLastNames = { first: "Ari", last: "Aran"};
4   const hasOnlyOne: FirstAndLastNames = { first: "Sappho" };
5
```

And also, we can't have excess properties.

```
2   type FirstAndLastNames = { first: string; last: string };
3   const hasBoth: FirstAndLastNames = { first: "Ari", last: "Aran"};
4   const hasThree: FirstAndLastNames = {
5                       first: "Sudhan",
6                       last: "Arvin",
7                       mid: "Ariel"
8               };
9
```

Note that excess property checks only trigger for object literals being created in locations that are declared to be an object type. Providing an existing object literal bypasses excess property checks.

NESTED OBJECT TYPES

We can nest objects in JS so for sure, we can nest types as shown below:

```
type Book = {
    author :  {
        name: string,
        age: number
    },
    name: string,
    year: number
}

const bookILoved: Book = {
    author: {
        name: "Vairamuthu",
        age: 71
    },
    name: "Kallikkaattu Ithigaasam",
    year: 2003
}
```

OPTIONAL PROPERTIES

We can include a ? before the : in a type property's type annotation to indicate that it's an optional property.

```
type Book = {
    name: string,
    year?: number
}

const bookILoved: Book = {
    name: "Kallikkaattu Ithigaasam"
}
```

When we include ? Before the colon of type annotation, the property actually takes undefined union the type provided. In this case, the type of year becomes "undefined | number". We can either assign a number or leave it!

OBJECT TYPE UNIONS

Why not? We can also union two object types.

```
type BookTypeOne = {
    name: string,
    year?: number
}

type BookTypeTwo = {
    name: string,
    year: number,
    authorName: string
}

const bookILoved: BookTypeOne | BookTypeTwo = {
    name: "Kallikkaattu Ithigaasam"
}
```

It means, We can also narrow it so...

```
const bookILoved: BookTypeOne | BookTypeTwo = Math.random() > 0.4 ?
{
    name: "Kallikkaattu Ithigaasam"
} : {
    name: "Kavithaigal",
    year: 2024,
    authorName: "Ariharasudhan"
}

if("year" in bookILoved){
    console.log(bookILoved.year);
}
```

DISCRIMINATED UNIONS

Another popular form of union typed objects in JavaScript and TypeScript is to have a property on the object indicate what shape the object is. This kind of type shape is called a discriminated union, and the property whose value indicates the object's type is a discriminant.

```
type BookTypeOne = {
    name: string,
    year?: number,
    type: "bookTypeOne"}

type BookTypeTwo = {
    name: string,
    year: number,
    authorName: string,
    type: "bookTypeTwo" }

const bookILoved: BookTypeOne | BookTypeTwo = Math.random() > 0.4 ?
{name: "Kallikkaattu Ithigaasam", type: "bookTypeOne" }:
{ name: "Kavithaigal", year: 2024, authorName: "Ariharasudhan",
  type: "bookTypeTwo"}

if(bookILoved.type === "bookTypeTwo"){
    console.log(bookILoved.year);
}
```

WE ALSO INTERSECT

What we want an object to follow multiple type definitions? We have something called, intersection. It is an and logic of types where union is an or logic.

```
type BookTypeOne = {
    name: string,
    year?: number,
}

type BookTypeTwo = {
    authorName: string
}

const bookILoved: BookTypeOne & BookTypeTwo = {
    name: "Ariyin Kavithaigal",
    year: 2024,
    authorName: "Ariharasudhan"
}
```

We can also combine intersection with union...

```
const bookILoved: BookTypeOne & BookTypeTwo | { lastname: string } = {
    name: "Ariyin Kavithaigal",
    year: 2024,
    authorName: "Ariharasudhan",
}
```

FUNCTION PARAMETERS

If we need a function to have parameters of types specified and to return a value of type specified, TypeScript says Yes!

```
2   function tellAboutMe(name: string, age: number): string {
3       return `You are ${name} and you are ${age} years old`;
4   }
5
6   tellAboutMe("Ari"); // age arg not provided
7   tellAboutMe("Ari", 21, 23); // Expected 2 but got 3
8   tellAboutMe(21, "Ari"); // not assignable
9   console.log(tellAboutMe("Ari", 21));
10  // You are Ari and you are 21 years old
11
```

We can have optional parameters as we could have optional properties.

```
2   function tellAboutMe(name: string, age?: number): string {
3       return `You are ${name} and you are ${age} years old`;
4   }
5
6   tellAboutMe("Ari"); // age arg not provided
7   console.log(tellAboutMe("Ari", 21));
```

If we want to have default values for parameters, remember! Type inference happens and once inferred, inferred!

```
function tellAboutMe(name: string, age = 21): string {
    return `You are ${name} and you are ${age} years old`;
}

tellAboutMe("Ari", 21);
tellAboutMe("Ari", 22);
tellAboutMe("Ari", "24"); // not assignable
```

We can also make it for rest parameters.

```
function tellAboutMe(name: string, ...ages: number[]): string {
    return `You are ${name} and you are ${ages[0]} years old`;
}

tellAboutMe("Ari", 21, 22);
tellAboutMe("Ari", 22);
tellAboutMe("Ari");

```

It doesn't end with these! Functions are the first class citizens!

We can pass them to other functions. So, we need to make the higher order functions, equipped with types!

```
2   const play = (num: number, name: string): string => {
3       return `${num} and ${name}!`
4   }
5
6   function tell(play: (num: number, name: string) => string ): string {
7       return play(21, "Hello");
8   }
9
10  console.log(tell(play)); // 21 and Hello!
```

Here, when we declare types for the function parameters, we use => to specify their return type. Beware of the paranthesis!

```
3   // Function return number or undefined value
4   let funcOne: () => number | undefined;
5   // Function returns a number or the function is undefined
6   let funcTwo: (() => number) | undefined;
```

Function Types can also be aliased as shown below.

```
type funcType = (index: number) => number;
const func: funcType = (index) => {
    return index*2;
}
```

A tip I can give is to declare the function with type and later define it as shown below.

```
let returner: (index: number) => number;

returner = (index: number) => {
    return "Hello";
}

// Type '(index: number) => string' is not assignable to type
// '(index: number) => number'.
//Type 'string' is not assignable to type 'number'.(2322)
```

When functions don't want to return anything, we can declare their return type as void.

```
function doNothing(): void {
    return;
}
function doNothingAgain(): void {
    return "Hello";
}
function doNothingAgainAgain(): void {
    console.log("ARI");
}
```

The never type is used to indicate that a function will never complete its execution successfully. This can happen for a variety of reasons, such as the function throwing an error, entering an infinite loop, or simply not returning a value at all.

```
function infiniteLoop(): never {
    while (true) {
        console.log("This will run forever!");
    }
}

function throwError(message: string): never {
    throw new Error(message);
}
```

If we want to perform or simulate something called Function Overloading in JavaScript,

We may need to check arguments.length and proceed accordingly. But, here in TypeScript, we can make it easy!

```
function heySiriCallAri(): void;
function heySiriCallAri(message: string): void;
function heySiriCallAri(contact: string, message: string): void;

function heySiriCallAri(contactOrMessage?: string, message?: string): void {
    if (typeof contactOrMessage === "undefined") {
        console.log("Calling Ari...");
    } else if (typeof message === "undefined") {
        console.log(`Calling Ari with message: ${contactOrMessage}`);
    } else {
        console.log(`Calling ${contactOrMessage} with message: ${message}`);
    }
}
heySiriCallAri();
heySiriCallAri("Hey Ari!");
heySiriCallAri("Ari", "Hello, how are you?");
```

TypeScript simplifies function overloading by allowing us to define multiple function signatures for the same function name. This enables us to specify different parameter types and return types.

ARRAYS

In JS, what we call as Array can hold any mixture of values.

```
3    const arr = ["Ari", 21, true];
4    arr.push("Arvin");
```

TypeScript respects the best practice of keeping to one data type per array by remembering what type of data is initially inside an array, and only allowing the array to operate on that kind of data. I mean, the type inference...

```
3    const arr = ["Ari", 21, true];
4    arr.push("Arvin");
5
6    //Argument of type 'undefined' is not assignable to
7    // parameter of type 'string | number | boolean'
8    arr.push(undefined);
9
```

The type annotation for an array requires the type of elements in the array followed by square brackets [ ].

```
let arr: number[];
arr = [1,2,3,4,5,6];
// Argument of type 'string' is not assignable to parameter of type 'number'.
arr.push("ARI");
arr.push(7);
// Type 'string' is not assignable to type 'number'
arr = [1,2,4,5, "Ari"];
```

I hope there is no perplexity!

```
// This function will return array of Strings
let stringsReturner: ()=> string[];
stringsReturner = ()=>{
    return ["Ari", "Haran", "Sudhan"];
}

// Array of Functions that return string
let functions: (()=> string)[];
functions = [()=>{return "Ari"}, ()=>{return "Haran"}];
```

We have to be careful with the paranthesis.

```
// Union Type : An array of strings or numbers
let arr: (string | number)[];
// Union Type : A string or An array of numbers
let arrTwo: string | number[];
arrTwo = "Ari";
arrTwo = [1,2,3,4];

```

Array types can be evolved. Yes! Every evolving array members! It is through using any.

```
let arr = [];
arr.push(21);    // number
arr.push("Ari"); // number || string
arr.push(true);  // number || string || boolean

```

We can define types for multidimensional arrays.

```
let arr: number[][]; // array of number arrays
arr = [
    [1,2],
    [3,4]
]
arr.push([1,3]);

let arr3D: number[][][]; // array of array of number arrays
arr3D = [
    [
        [1,2],
        [3,4]
    ],
    [
        [1,2],
        [3,4]
    ],
]
arr3D.push([[1,3], [3,4]]);
```

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

```
let arr: number[];
arr = [1,2,3];
let num = arr[0];
num = "Ari"; // Type 'string' is not assignable to type 'number'.

let arrUnion: (number | string)[];
arrUnion = [1,2,"Ari"];
let unnum = arrUnion[0];
let unstr = arrUnion[2];
unstr = 23;
unnum = "Ari";
```

But, there is actually one caveat. TS infers the type of the elements to be what it was defined as. Can we access an element using index beyond the length of the array?

Will it be undefined? No! Instead, it will be the type defined.

```
let arr: number[];
arr = [1, 2, 3];
let num = arr[1234]; // type is number not undefined
```

Arrays can be joined together using the spread (...) operator. In TS, if the input arrays are of same type, the output array will be that same type. If the input arrays are of different types, the output array will be the union of those types.

```
let nArr: number[];
let sArr: string[];

nArr = [1,2,3];
let combined = [...nArr, "Ari", "Ariel"]; // number | string
combined.push(23);
combined.push("Hello");
combined.push(true);
```

TS recognizes and will perform type checking on the JavaScript practice of ... spreading an array as a rest parameter. Arrays used as arguments for rest parameters must have the same array type as the rest parameter.

```
2   const func = (num: number, ...names: string[]) => {
3
4   }
5   func(2, "Ari", "Hello"); // rest is of type string
6
```

TUPLES

In JS, what we call as Array can hold any mixture of values and it shrinks and expand over the time. If we want to make it fixed in terms of size and the order of types, we can use something called Tuple.

```
let tup: [number, string];
tup = [1, "Ari"];
tup = ["Ari", 1]; // ORDER MATTERS
tup = [1, "Ari", 2]; // Source has 3 element(s) but target allows only 2.
tup[0] = 23;
tup[0] = "Ari"; // Type 'string' is not assignable to type 'number'

```

We should be careful with inferred arrays. Although they look similar morphologically, they will get a union type.

```
const pairLoose = [false, 123]; // Array: (boolean | number)[]
const pairTupleLoose: [boolean, number] = pairLoose;
// Type '(number | boolean)[]' is not assignable to type '[boolean, number]'.
// Target requires 2 element(s) but source may have fewer.
```

What if we use tuples as rest parameters?

```
const myFunc = (num: number, str: string) => {
    arr[0] = "Hello";
}

let myTupleCorrect: [number, string] = [21, "Ari"];
let myTupleInCorrect: [string, number] = ["Ari", 21];
myFunc(...myTupleCorrect);
// Argument of type 'string' is not assignable
// to parameter of type 'number'
myFunc(...myTupleInCorrect);
```

If we check the variables which stores the values returned from a function as an array, the variables will have a union (of all return types) type.

```
function myFunc(){
    return [1,"Ari"];
}

let [num, nameOfHim] = myFunc(); // num and nameOfHim: string | number
num = "Ari";
nameOfHim = 21;
```

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.

```
function myFunc(): [number, string]{
    return [1,"Ari"];
}

let [num, nameOfHim] = myFunc(); // num and nameOfHim: string | number
num = "Ari"; // Type 'string' is not assignable to type 'number'
nameOfHim = 21; // Type 'number' is not assignable to type 'string'

```

If we want to convert an array into a read-only tuple, we can do that with const assertion.

```
let myMeta = ["Ari", 21] as const;
// Cannot assign to '0' because it is a read-only property
myMeta[0] = "Hello";
```

With this, we don’t need to specify the type explicitly for creating tuple. But, remember that, it becomes immutable!

INTERFACES

We can make our own types. It's not like aliasing. A type alias is a way to create a new name for any type, including primitives, unions, intersections, and other types like functions and tuples. An interface is a way to define the shape of an object, focusing primarily on object structure. It is more flexible when working with object-oriented concepts and supports declaration merging. Interfaces can "merge" together to be augmented and can be used to type check the structure of class declarations while type aliases cannot.

We'll see these later. So, let's define an interface now.

```
2    interface Animal {
3        name: string,
4        scname: string,
5        weight: number
6    }
7
8    const tiger: Animal = {
9        name: "Tiger",
10       scname: "panthera leo",
11       weight: 234
12   }
```

As in type aliases, we can't assign a wrong typed value.

```
7    const tiger: Animal = {
8        name: 21,
9        scname: "panthera leo",
10       weight: 234
11   }
12   const cheetah: Animal = "Ari";
```

We can also have optional properties as well.

```
interface Animal {
    name: string,
    scname?: string,
    weight: number
}
const tiger: Animal = {
    name: "tiger",
    weight: 234
}
```

Besides, we can make a property read-only using readonly modifier.

```
interface Animal {
    readonly name: string,
}
const tiger: Animal = {
    name: "tiger"
}
// Cannot assign to 'name' because it is a read-only property
tiger.name = "lion";
```

It’s very common in JavaScript for object members to be functions. TypeScript therefore allows declaring interface members as being the function types.

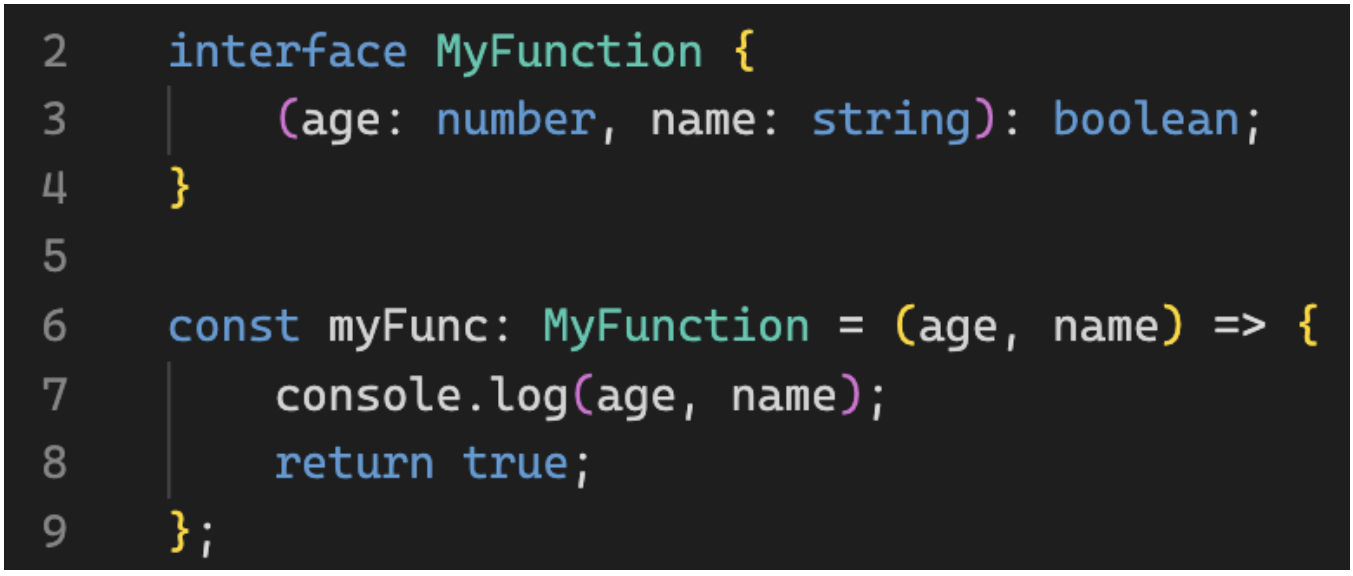
```
interface Mine {
    prop: () => string;
    method(): string;
};

const hasBoth: Mine = {
    prop: () => "",
    method() {
        return "";
    }
};
```

Both parameters can be optional.

```
interface Mine {
    prop?: () => string;
    method?() : string;
};
```

A call signature in an interface is a way to define how a function should be structured—what parameters it takes and what type it returns. Interface defines a structure or contract for an object. Meanwhile, A Call Signature defines how a function should be called, including its parameters and return type.



```
interface MyFunction {
    (age: number, name: string): boolean;
}

const myFunc: MyFunction = (age, name) => {
    console.log(age, name);
    return true;
};
```

Index signatures allow us to define the types of keys and their corresponding values in an object, without knowing the exact names of the keys upfront.

```
interface MyDictionary {
  [key: string]: string;
}

const tamizhWords: MyDictionary = {
  "Ari": "Lion",
  "Vellam": "Jaggery"
};

const malayalamWords: MyDictionary = {
  "Ari": "Rice",
  "Vellam": "Water"
};
```

Remember! It is also possible with type aliasing.

```
type Dictionary = {
  [key: string]: number;
};
```

We cannot mix different types of values like number with an index signature like [key: string]: string. If we want to mix, mix with same type! Don't mess with other types!

```
interface MyDictionary {
  [key: string]: string,
  year: number //Property 'year' of type 'number' is
               // not assignable to 'string' index type 'string'
}

interface YourDictionary {
  [key: string]: string,
  year: string
}
```

Woh! We can also have some literal types.

```
interface Ages {
  [key: string]: number,
  year: 2024
}
const Ari: Ages = {
    age: 21,
    year: 2024
}
```

We can’t leave the literal property and assign values other than defined.

```
10    const Jonathan: Ages = {
11        age: 191,
12        year: 2023
13    }
```

We can nest interfaces within.

```
2     interface College {
3         college: string,
4         studentDetails: Student
5     }
6
7     interface Student {
8         name: string,
9         age: number
10    }
```

TypeScript allows an interface to extend another interface, which declares it as copying all the members of another.

```
interface IHaveNumber {
    age: number
}
interface IHaveBoolAndString extends IHaveNumber {
    isMarried: boolean,
    name: string
}

const Ari : IHaveBoolAndString = {
    age: 21,
    name: "Ari",
    isMarried: false
}
```

Types can be over-ridden while extending but remember the type in base interface must be compatible with child interface.

TypeScript uses structural typing, so when we extend Person, it expects Employee to still have an age property of type of string or number.

```
interface Person {
  name: string;
  age: string | number;
}

interface Employee extends Person {
  age: number;       // Override the `age` property with a new type
  position: string;
}

const Emp: Employee = {
    name: "Ari",
    age: 21,      // age: "someStr" would be an error
    position: "TechStaff"
}
```

We can extend multiple interfaces. This means that an interface can inherit properties and methods from more than one interface.

We combine the structure of multiple interfaces into one.

```
interface Person {
    pMethod(): number
}

interface Employee {
    eMethod(): boolean
}

interface Alien extends Person, Employee {
    aMethod(): string
}

const Mixed: Alien = {
    pMethod() { return 1 },
    eMethod() { return true },
    aMethod() { return "Alien" }
}
```

Interface merging occurs when we declare the same interface multiple times (Subsequent property declarations).

TypeScript automatically merges these declarations into a single interface, combining their properties. This is particularly useful for extending existing interfaces without modifying the original declaration directly.

```
interface Person {
    pMethod(): number,
    name: string
}

interface Person {
    eMethod(): boolean,
}

const Mixed: Person = {
    pMethod() { return 1 },
    eMethod() { return true },
    name: "Ari"
}
```

Note: Subsequent property declarations must have the same type!

```
interface Person {
    pMethod(): number,
    name: string
}

interface Person {
    name: number; //Subsequent property declarations must have the same type.
}
```

When we merge two interfaces that have a method with the same name but different parameter types, TypeScript creates overloads.

```
interface Logger {
  log(message: string): void; // Method that takes a string
}

interface Logger {
  log(message: number): void; // Method that takes a number
}

const logger: Logger = {
  log: (message) => {
    console.log(`Log: ${message}`);
  },
};

logger.log("Hello, world!");
logger.log(34);
```

CLASSES

Classes are blueprints for creating objects. They encapsulate data (properties) and behavior (methods) into a single entity, enabling Object-Oriented Programming principles.

```
3   class Person {
4     name: string;
5     age: number;
6     constructor(name: string, age: number) {
7       this.name = name;
8       this.age = age;
9     }
10    greet() {
11      console.log(`Hello, my name is ${this.name}`);
12    }
13  }
14
15  new Person("Ari", 21).greet();
16  // Hello, my name is Ari
```

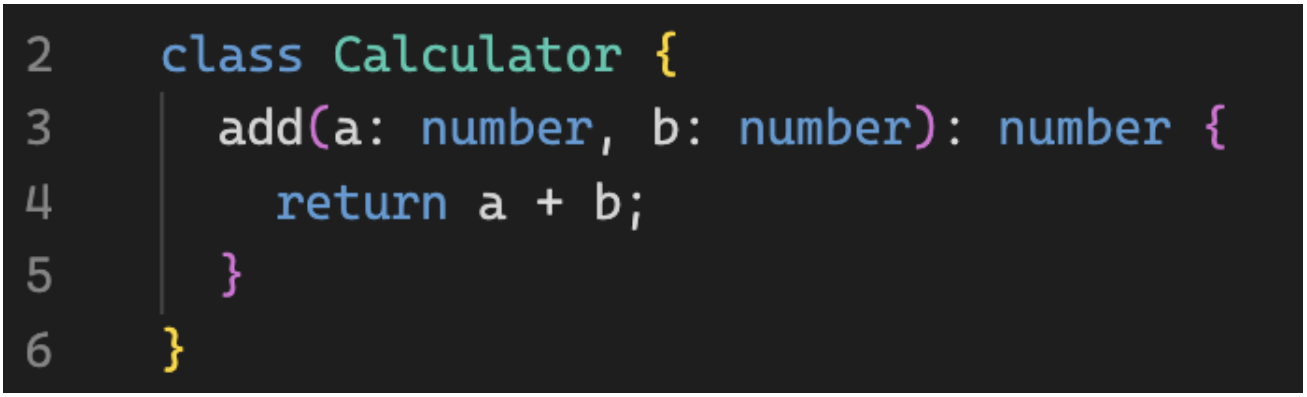
Class properties are variables that belong to a class. They define the state of an object created from the class.

```
class Bird {
  name: string;
  scname: string;
  weight: number;
  constructor(name: string, scname: string, weight: number) {
    this.name = name;
    this.scname = scname;
    this.weight = weight;
  }
}

const bird = new Bird("Peacock", "Pavo cristatus", 6);
const nameOfBird: string = bird.name;
const scName: string = bird.scname;
const weight: number = bird.weight;
// Property 'height' does not exist on type 'Bird'
const height: number = bird.height;
```

TypeScript checks that class properties are properly initialized before they are used. If a property is not assigned a value in the constructor, TypeScript will raise an error.

Function properties are methods defined within a class that can be called on instances of that class. They define behaviours associated with the class.



Although strict initialization checking is useful most of the time, we may come across some cases where a class property is intentionally able to be unassigned after the class constructor.

Definitely Assigned Properties are properties that are guaranteed to be initialized when the class is instantiated. We can use the ! (definite assignment assertion) to indicate that the property will be initialized later.

```
class User {
  name!: string; // Definite assignment assertion
  constructor() {
    // name will be assigned later
  }
}
```

You know what... Optional properties can also exist! Optional properties are defined with a ? as we know already.

```
class User {
  name?: string;
}
```

Read-only properties can only be assigned a value at the time of declaration or in the constructor. After that, their value cannot be changed.

```
class User {
  readonly name: string;
  constructor(name: string) {
    this.name = name;
  }
}

new User("Ari").name = "Haran";
// Cannot assign to 'name' because
// it is a read-only property.
```

Classes can be used as types in TypeScript.

We can create objects that adhere to the class structure, allowing for type checking.

```
2   class Person {
3     name: string;
4     age: number;
5
6     constructor(name: string, age: number) {
7       this.name = name;
8       this.age = age;
9     }
10  }
11
12  const person: Person = new Person("Ari", 21);
```

Interfaces can describe the shape of a class, allowing for the definition of expected properties and methods. A class can implement an interface, enforcing that it adheres to the interface structure.

Look at the following example.

```
interface Shape {
  area(): number;
}

class Rectangle implements Shape {
  constructor(private width: number, private height: number) {}
  area(): number {
    return this.width * this.height;
  }
}
```

A class can implement multiple interfaces, allowing it to adhere to multiple structures.

```
interface Shape {
  area(): number;
}

interface Dummy {
  dummy(): string;
}

class Rectangle implements Shape, Dummy {
  constructor(private width: number, private height: number) {}
  area(): number {
    return this.width * this.height;
  }
  dummy(): string {
    return `${this.height}`;
  }
}
```

A class can extend another class, inheriting its properties and methods. This promotes code reuse and a hierarchical structure.

```
2   class Animal {
3     speak() {
4       console.log("Animal speaks");
5     }
6   }
7   class Dog extends Animal {
8     bark() {
9       console.log("Dog barks");
10    }
11  }
12
13  new Dog().speak();
```

When assigning an object of a derived class to a variable of the base class type, TypeScript checks that the derived class is compatible with the base class.

```
13    let animal: Animal;
14    animal = new Dog();
```

When a derived class overrides the constructor of a base class, it can call the base class constructor using super().

```
class Child extends Parent{
  constructor( ) {
    super( );
    }
 }
```

A derived class can override methods from its base class, providing a specific implementation.

```
class Parent {
    method( ) {
        console.log(“Parent”);
    }
}
class Child extends Parent{
    method( ) {
        console.log(“Child”);
    }
}
```

When overriding properties in a derived class, the property must match the type of the base class property.

```
class Parent {
    message: string = "Hello";
}
class Child extends Parent{
    message: string = "Hi";
}
```

Abstract classes cannot be instantiated directly. They can define abstract methods that must be implemented by derived classes.

TypeScript provides access modifiers for class members: public, private, and protected.

public: Accessible from anywhere.

private: Accessible only within the class

protected: Accessible within the class and subclasses.

```
2    class Example {
3      public name!: string;
4      private age!: number;
5      protected id!: number;
6    }
```

Static members belong to the class itself rather than to instances of the class. They are accessed using the class name.

```
2    class Counter {
3      static count: number = 0;
4      static increment() {
5        Counter.count++;
6      }
7    }
8
9    Counter.increment();
10   console.log(Counter.count); // Output: 1
11   Counter.increment();
12   console.log(Counter.count); // Output: 2
```

TYPE MODIFIERS

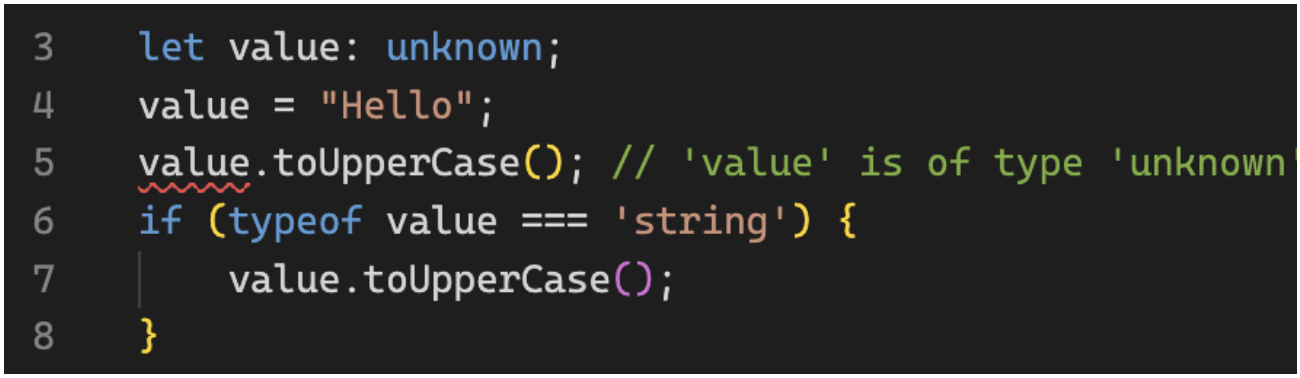
There are types that can represent any value. Such types are called, **Top Types**. The most common top type in TypeScript is **any**. It can hold values of any type, allowing maximum flexibility but at the cost of type safety.

```
2    let value: any;
3    value = 42;          // Valid
4    value = "Hello";     // Valid
5    value = true;        // Valid
```

The following will surely crash during the run time.

```
3    let value: any;
4    value = 32;
5    value.toUpperCase();
```

The **unknown** type is safer than any. We can assign any value to it, but before performing operations on an unknown type, we need to assert its type or narrow it down. This provides better type safety than any.



The **keyof** operator creates a union of string literal types representing the keys of an object type.

It helps in creating types that depend on the keys of another type.

```
interface Person {
  name: string;
  age: number;
}
const MyPerson: Person = {
    name: "Ari",
    age: 21
}
type PersonKeys = keyof Person; // "name" | "age"

const myPersonAge: PersonKeys = "age";
const myPersonName: PersonKeys = "name";
```

The typeof operator is used to get the type of a variable or object at compile time. It can also be used to create types based on existing variable types.

```
const MyPerson = {
    name: "Ari",
    age: 21
}
```

```
type PersonType = typeof MyPerson;
// { name: string; age: number; }
const YourPerson: PersonType = {
    name: "Ariel",
    age: 22
}
```

The combination of keyof and typeof allows us to create a union of the keys of a variable's type.

```
const MyPerson = {
    name: "Ari",
    age: 21
}
type PersonKey = keyof typeof MyPerson;
const nameKey: PersonKey = "name";
const ageKey: PersonKey = "age";
```

Type assertions are to tell the TypeScript compiler to treat a variable as a specific type, overriding the type inference.

We can use the **as** keyword or angle-bracket syntax.

```
let someValue: unknown = "Hello";
let strLength: number = (someValue as string).length; // Using 'as'
```

Assertions change the type of an existing variable.

Const assertions tell TypeScript to infer a more specific, immutable type for a value.

```
const colors = ["red", "green", "blue"] as const;
// colors is of type readonly ["red", "green", "blue"]
```

The bottom type refers to the type that represents no value or never occurs. This type is denoted by the keyword **never.**

```
3   function throwError(message: string): never {
4     throw new Error(message);
5   }
6   function infiniteLoop(): never {
7     while (true) {}
8   }
```

GENERICS

Generics are used to define a component like a function, class, or interface that can work with multiple types rather than a single one. We can think of it as a way to pass types as parameters to components. This increases code flexibility and reusability.

Look at the following example which uses Generics.

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

Here, <T> is a generic type parameter.

The function identity can take any type T, and it ensures the input and return types are the same. Interested? Let's write an example.

```
function identity<T>(arg: T): T {
  return arg;
}
const str = identity<string>("Hello");
```

We can have multiple type parameters for more complex functions.

```
function combine<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}
combine("Ari", true);
```

We can also make an interface generic.

Look at the following example.

```
interface Boy<S, I> {
    name: S,
    age: I
}

let SouthernBoy: Boy<string, number>;
SouthernBoy = {
    name: "Ari",
    age: 21
}
```

What about making classes Generic?

```
class GenericBoy<T, U> {
  name: T;
  age: U;
  constructor(name: T, age: U) {
    this.name = name;
    this.age = age;
  }
}
new GenericBoy<string, number>("Ari", 21);
```

Let’s extend a generic class!

```
class GenericBoy<T, U> {
  name: T;
  age: U;
  constructor(name: T, age: U) {
    this.name = name;
    this.age = age;
  }
}
class SouthernBoy<T, U> extends GenericBoy<T, U> {
  constructor(name: T, age: U) {
    super(name, age);
  }
}
const boy = new SouthernBoy<string, number>("Ari", 21);
console.log(boy);
```

It becomes interesting when we implement a generic interface.

```
interface Container<T> {
  getItem: () => T;
}

class StringContainer implements Container<string> {
  getItem() {
    return "Item";
  }
}
```

Methods in a class can also be generic.

```
class Utility {
  static combine<T, U>(a: T, b: U): [T, U] {
    return [a, b];
  }
}
```

We can create generic type aliases!

```
type Pair<T, U> = [T, U];
let pairOne: Pair<string, number> = ["A", 1];
let pairTwo: Pair<number, number> = [21, 1];
```

We can also apply the readonly, optional to generics. Just as default values in methods, we can also declare the default type in Generics.

```
type Pair<T, U=number> = [T, U];
let pairOne: Pair<string, number> = ["A", 1];
let pairTwo: Pair<number> = [21, 1];
```

We can constrain generics to extend a specific type.

```
function printName<T extends { name: string }>(person: T): void {
  console.log(person.name);
}

const person = { name: "Ari", age: 25 };
printName(person);

const invalidPerson = { age: 25 };
printName(invalidPerson);
// Argument of type '{ age: number; }' is not assignable to parameter
// of type '{ name: string; }'. Property 'name' is missing in
// type '{ age: number; }' but required in type '{ name: string; }'
```

keyof with generics lets you constrain parameters to valid object keys.

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
```

Promises are not exceptions to TypeScript!

```
const promise: Promise<string> = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve("Task completed successfully!");
    } else {
      reject("Task failed!");
    }
  }, 2000);
});
```

```
promise
  .then((result: string) => {
    console.log(result); // Output: Task completed successfully!
  })
  .catch((error: string) => {
    console.error(error); // If rejected, it will output: Task failed!
  });
```

The Promise.reject(err) function, like the Promise.resolve method, always returns a rejected promise. The promise rejection value is derived from the error parameter, and its type is any. The **async and await** provide a cleaner, more readable way to work with promises in TypeScript. An async function always returns a promise, and you use await to pause the execution until the promise is resolved or rejected.

```typescript
async function fetchData(): Promise<string> {
  const data = await new Promise<string>((resolve) => {
    setTimeout(() => resolve("Fetched Data"), 1000);
  });
  return data;
}
async function main() {
  try {
    const result = await fetchData();
    console.log("Result:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}
main();
```

Promises can be typed for better type safety!

NANDRI