

# INTRODUCTION TO TYPESCRIPT



# THIS BOOK

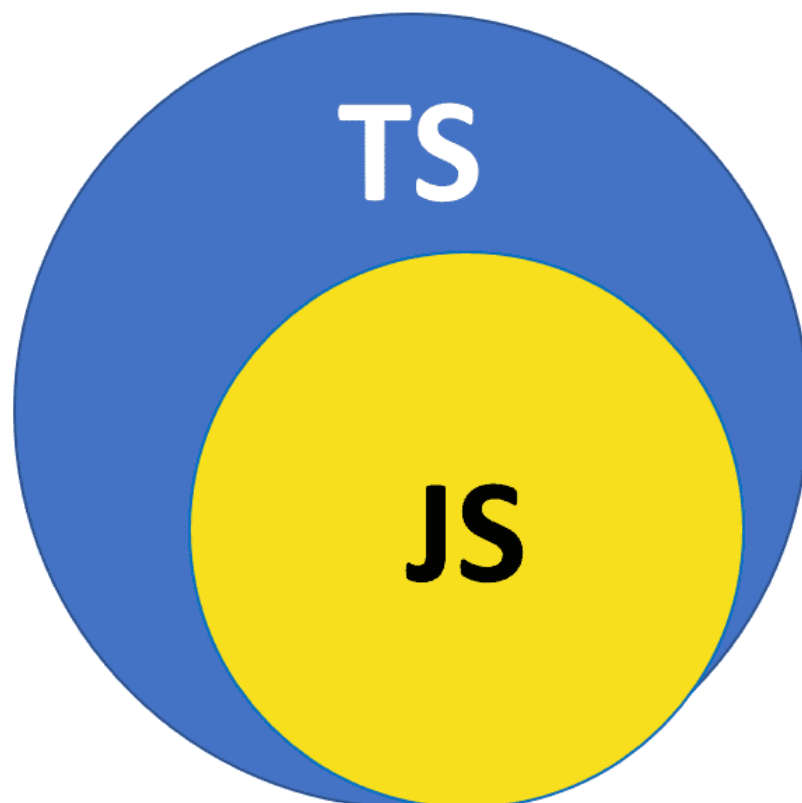
This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

<https://arihara-sudhan.github.io>



# Install & Transpile & Compile

JavaScript is loosely typed. That means, variables are not bound to a specific data type at the time of declaration. We can assign any type of data to the variable. Meanwhile, TypeScript is a strongly typed language which gives The Power of Types. With TypeScript, We can extend JavaScript by adding types.

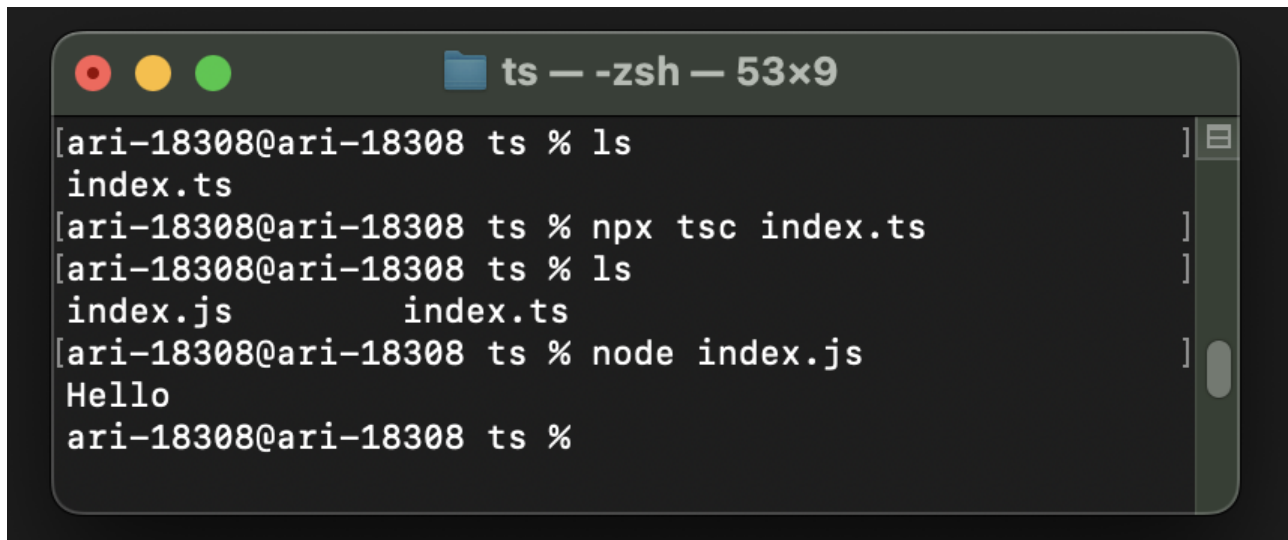


Using types has its own advantages. Evidently, We can get a **good error feedback!** In TypeScript, we have the option for defining **our own types** as in other languages such as Java. A simple definition for TypeScript is **JavaScript with syntax for types**. It can be easily installed using node package manager. Let's write a ts file and transpile it into JavaScript using `npx tsc` command. Let's write a ts file which logs "Hello". After that, let's transpile it using **`npx tsc filename.js`**.

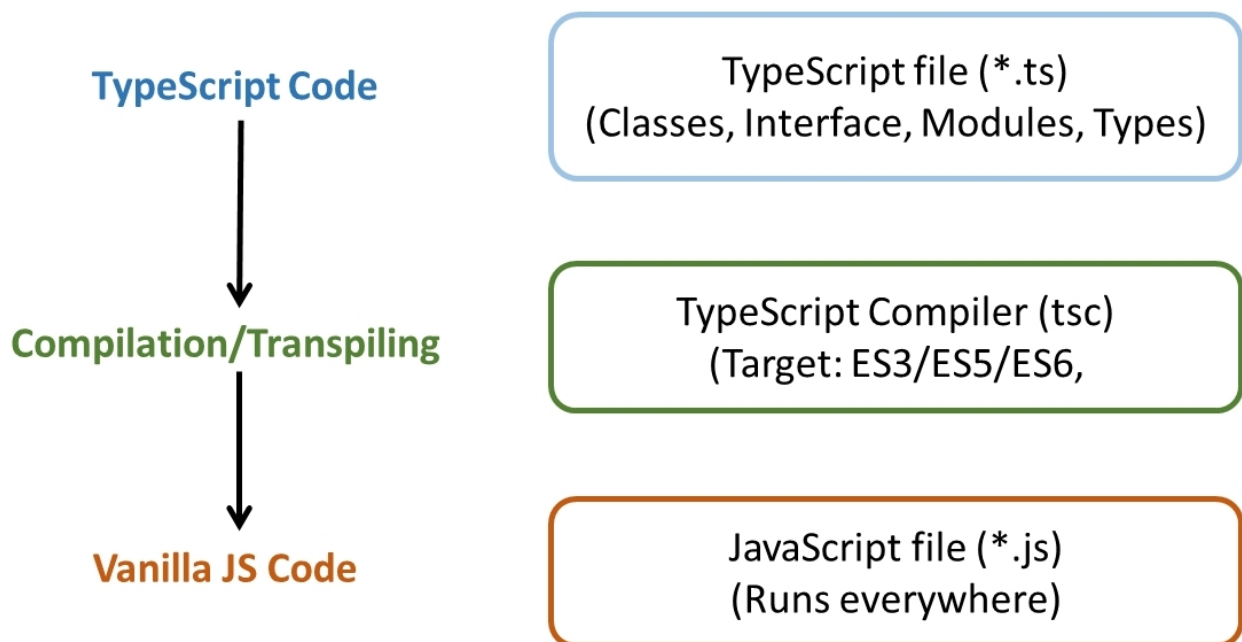
```
TS index.ts
```

```
1 console.log("Hello");
```

When we transpile it, there will be an equivalent JavaScript file created, which can be executed using node command.



```
ts — -zsh — 53x9
[ari-18308@ari-18308 ts % ls
index.ts
[ari-18308@ari-18308 ts % npx tsc index.ts
[ari-18308@ari-18308 ts % ls
index.js      index.ts
[ari-18308@ari-18308 ts % node index.js
Hello
ari-18308@ari-18308 ts %
```



# Type Basics

We can explicitly type a variable using the type annotation symbol colon.

```
TS index.ts > ...
1  let ageOfHim: number = 21.76;
2  let nameOfHim: string = "Ari";
3  let lifePartner: undefined = undefined;
4  let badHabit: null = null;
5  let isHeEligible: boolean;
6  isHeEligible = true;
7
8  isHeEligible = 21; // Type 'number' is not assignable to type 'boolean'
9
```

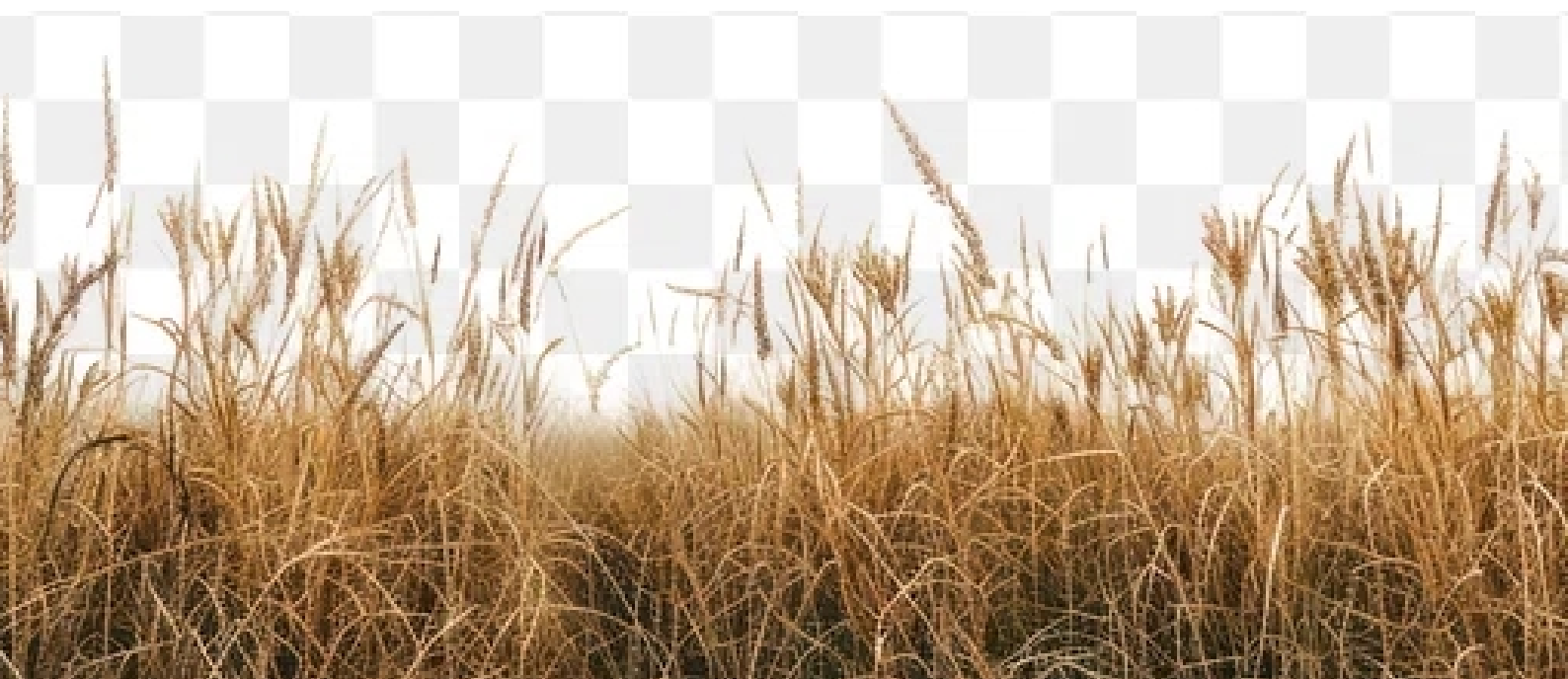
As we can note here, we can't assign a data of types other than the defined. This makes JavaScript, harnessed of the strong typing feature. Now, let's transpile it after removing the 8<sup>th</sup> line. If we look into the index.js file generated, we may be stumbled upon it. When we transpile TypeScript to JavaScript, the TypeScript compiler (tsc) removes the type annotations because JavaScript



does not support types. TypeScript's type annotations are used during development for type checking and do not carry over to the compiled JavaScript code.

```
JS index.js > ...
```

```
1   var ageOfHim = 21.76;  
2   var nameOfHim = "Ari";  
3   var lifePartner = undefined;  
4   var badHabit = null;  
5   var isHeEligible;  
6   isHeEligible = true;
```



# Configuration File

To simplify the process of transpiling, we can use a configuration file. We can create such a configuration file using `npx tsc --init` command.

```
[ari-18308@ari-18308 ts % npx tsc --init  
  
Created a new tsconfig.json with:  
  
  target: es2016  
  module: commonjs  
  strict: true  
  esModuleInterop: true  
  skipLibCheck: true  
  forceConsistentCasingInFileNames: true  
  
You can learn more at https://aka.ms/tsconfig  
ari-18308@ari-18308 ts %
```

Now, there will be a configuration file generated. We just have to specify whatever we want to.



I would like to set the rootDir where ts files should be present and also outDir where transpiled js files should reside. Just tell it!

```
tsconfig.json > {} compilerOptions
2  "compilerOptions": {
29    "rootDir": "./src",                /* Specify the root folder within your source
30    // "moduleResolution": "node10",    /* Specify how TypeScript looks up a file fro
31    // "baseUrl": "./",                 /* Specify the base directory to resolve non-
32    // "paths": {},                      /* Specify a set of entries that re-map impor
33    // "rootDirs": [],                  /* Allow multiple folders to be treated as on
34    // "typeRoots": [],                 /* Specify multiple folders that act like './
35    // "types": [],                     /* Specify type package names to be included
36    // "allowUmdGlobalAccess": true,    /* Allow accessing UMD globals from modules.
37    // "moduleSuffixes": [],            /* List of file name suffixes to search when
38    // "allowImportingTsExtensions": true, /* Allow imports to include TypeScript file e
39    // "resolvePackageJsonExports": true, /* Use the package.json 'exports' field when
40    // "resolvePackageJsonImports": true, /* Use the package.json 'imports' field when
41    // "customConditions": [],          /* Conditions to set in addition to the resol
42    // "resolveJsonModule": true,       /* Enable importing .json files. */
43    // "allowArbitraryExtensions": true, /* Enable importing files with any extension,
44    // "noResolve": true,                /* Disallow 'import's, 'require's or '<refere
45
46    /* JavaScript Support */
47    // "allowJs": true,                  /* Allow JavaScript files to be a part of you
48    // "checkJs": true,                  /* Enable error reporting in type-checked Jav
49    // "maxNodeModuleJsDepth": 1,        /* Specify the maximum folder depth used for
50
51    /* Emit */
52    // "declaration": true,              /* Generate .d.ts files from TypeScript and J
53    // "declarationMap": true,           /* Create sourcemaps for d.ts files. */
54    // "emitDeclarationOnly": true,      /* Only output d.ts files and not JavaScript
55    // "sourceMap": true,                /* Create source map files for emitted JavaSc
56    // "inlineSourceMap": true,          /* Include sourcemap files inside the emitted
57    // "outFile": "./",                  /* Specify a file that bundles all outputs in
58    "outDir": "./dist",                 /* Specify an output folder for all emitted
59    // "removeComments": true,          /* Disable emitting comments. */
60    // "noEmit": true,                  /* Disable emitting files from a compilation.
```

Now, we just need to type npx tsc. It will compile and the transpiled code will be there in dist folder as we specified.

✓ TS

✓ dist

JS index.js

✓ src

TS index.ts

TS tsconfig.json

We don't need to manually transpile it each time we change the ts file in src folder. We can simply use the `npx tsc --watch` command which will watch for changes in ts file and transpile the code immediately if any.

## Type Basics II

If we want to specify the type to be array of any types, we can do that as in the following.

```
src > TS index.ts > ...  
1   let age: number[];  
2   age = [1,2,3,4,5,"Ari"]
```

Now, we can only keep numbers in the array. It gives WOW benefits while development.

```
src > TS index.ts > ...  
1   let age: number[];  
2   age = [1,2,3,4,5];  
3  
4   age.push(23);  
5   age.push("Hello");
```

Type can be inferred in arrays. For example, the following snippet doesn't explicitly tell the type but, we couldn't push a value of another type.

```
src > TS index.ts > ...  
1   let fruits = ["Apple", "Orange"];  
2   fruits.push(23);  
3
```

The array becomes list if we have heterogeneous data in it. It is inferred thus.

```
src > TS index.ts > ...  
1   let fruits = ["Apple", "Orange", 23];  
2   fruits.push(23);  
3
```

It becomes even more interesting when we look at type annotation of objects.

```
src > TS index.ts > ...  
1   let person : {name: string, age: number} = {  
2       name: "Ari",  
3       age: 21  
4   }
```

A property which wasn't typed, can't be added in.

```
src > TS index.ts > ...
1   let person : {name: string, age: number} = {
2   |     name: "Ari",
3   |     age: 21
4   |   }
5
6   person.place = "Nellai"; //Property 'place' does not exist on type '{ name: string; age: number; }'
7
```

The type inference will also happen for the objects.

```
src > TS index.ts > ...
1   let person = {
2   |     name: "Ari",
3   |     age: 21
4   |   }
5   (property) name: string
6   person.name = "Ari";
7
8
```

Remember! In a TypeScript file, We can't add new properties unless we redefine our object.

We can define types for functions too.

```
src > TS index.ts > ...  
1   function myFunc(num: number): number {  
2   |       return num*num;  
3   }
```

Return type is specified at the end of the function name and () as shown above. The types of parameters are defined as that of a normal variable.

The same goes for Arrow Functions!

```
src > TS index.ts > [🔍] myFunc  
1   const myFunc = (num: number): number => {  
2   |       return num*num;  
3   |   }  
4
```

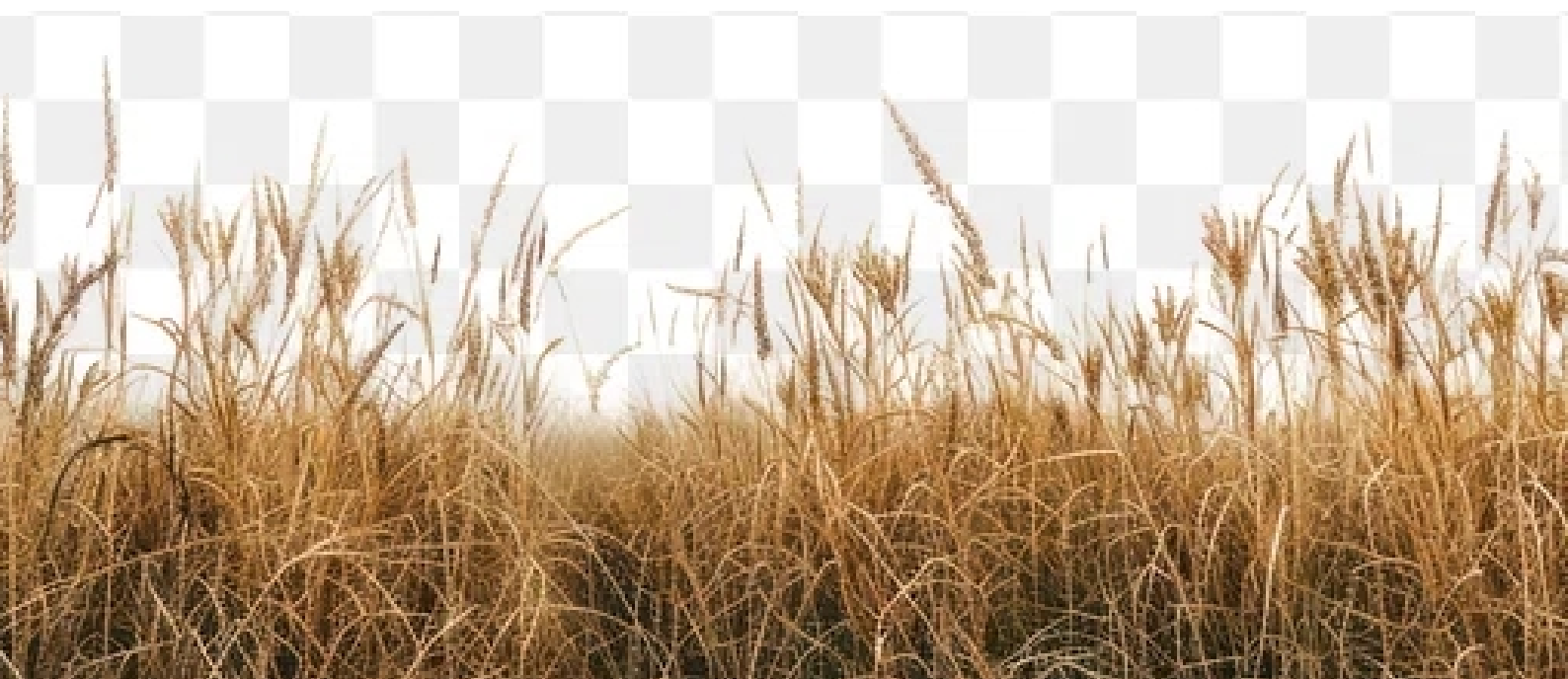
If I try to return a data of type other than number, it will throw an error. And also, if we pass an argument of type other than the number, it will throw an error.



```
src > TS index.ts > ...
1  const myFunc = (num: number): number => {
2    |    return num*num;
3  }
4
5  Argument of type 'string' is not assignable to parameter of type
6  'number'. ts(2345)
7
8  View Problem (⌘F8) No quick fixes available
9
10 myFunc("Ari");
```

There happens an inference for the return type of the function.

```
src > TS index.ts > ...
1
2  💡
3
4  const myFunc: (num: number) => number
5
6  const myFunc = (num: number) => {
7    |    return num*num;
8  }
9
10 myFunc(2);
```



# AnyType

There is a specifier, “any” by means of which, we can define a data of any type.

```
src > TS index.ts > ...  
1   let num: any;  
2   num = 21;  
3   num = {  
4       |   name: "Ari",  
5       |   age: 21  
6   }
```

An array of any type can also be defined.

```
src > TS index.ts > ...  
1   const myFunc = (nums: any[]): any[] => {  
2       |   return nums.map(num=> (num*num))  
3   }  
4  
5   myFunc([1, 2, 2.45]);  
6
```

# Type: Tuple

Tuple, can be defined as an array of data of different datatypes. It can be defined as the following:

```
src > TS index.ts > ...
```

```
1   const myTupe: [string, number, boolean] = ["Ari", 21, true];
```

So, a good definition can be, “Tuples are a type of array with a fixed number of elements where each element can have a different type”. They are particularly useful when we need to represent a fixed structure with multiple pieces of related data. Tuples provide a way to work with heterogeneous data and ensure that the data follows a specific structure/order. We can also define an optional parameter using ? Mark.

```
src > TS index.ts > ...  
1   const myTup: [string, number, boolean?] = ["Ari", 21];
```

We can use this when we want to return multiple value of different types from a function.

```
src > TS index.ts > ...  
1   const myFunc = ():[string, number] => {  
2   |       return ["Ari", 21];  
3   |   }  
4  
5   const [whoIAM, age] = myFunc();
```

If we want to specify, for which data a particular type is defined for, we can use a named tuple.

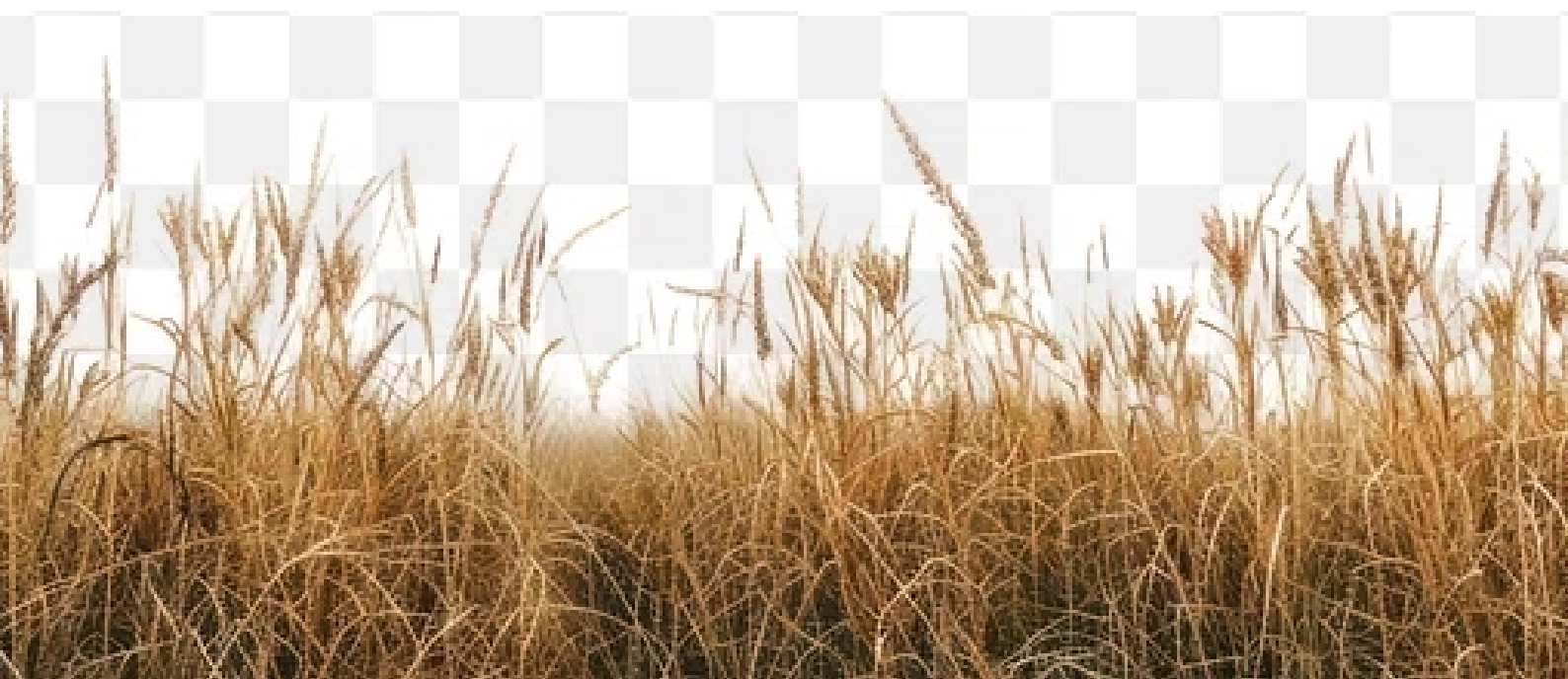
```
src > TS index.ts > ...  
1   let myTuple: [myName: string, age: number];  
2  
3  
4  
5   let myTuple: [myName: string, age: number]  
6   myTuple = ["Ari", 21];
```

# Interface: Custom Types

In TypeScript, an interface is a powerful way to define the shape of an object, describing the structure that an object must adhere to. Interfaces allow us to specify the types of properties and methods an object should have, making our code more predictable and maintainable.

```
src > TS index.ts > ...  
1   interface User {  
2       myName: string,  
3       age: number,  
4       tellName(firstName: string): string,  
5       weight?: number  
6   }  
7
```

```
8   const myObj: User = {
9     myName: "Ari",
10    age: 21,
11    tellName: (firstName)=>{
12      |    return firstName
13    }
14  }
15
16
17  const myObjTwo: User = {
18    myName: "Ariharan",
19    age: 22,
20    tellName: (firstName)=>{
21      |    return firstName
22    }
23  }
24
25  const users: User[] = [myObj, myObjTwo]
```





Interface supports to be extended by another interface and to be implemented by a class.

```
src > TS index.ts > ...
1  interface Person {
2      name: string;
3      age: number;
4  }
5
6  interface Employee extends Person {
7      employeeId: number;
8  }
9
10 const employee: Employee = {
11     name: "Eve",
12     age: 28,
13     employeeId: 1234,
14 };
15
```

Classes can implement interfaces, enforcing that the class conforms to the interface structure.

```
src > TS index.ts > ...
1  interface Animal {
2      name: string;
3      speak(): void;
4  }
5
6  class Dog implements Animal {
7      name: string;
8
9      constructor(name: string) {
10         this.name = name;
11     }
12     speak() {
13         console.log(`${this.name} barks.`);
14     }
15 }
16
17 const myDog = new Dog("Buddy");
18 myDog.speak();
19
```

# Type Aliases

Apart from the usage of Interfaces, another way to make custom types is to use the type alias.

```
src > TS index.ts > ...
1   type RGBA = [number, number, number, number];
2
3   function giveRGBA(): RGBA {
4       |   return [255,255,255,0.5];
5   }
6
```

We can easily define a custom type!

```
src > TS index.ts > ...
1   type USER = {
2       |   userName: string,
3       |   userAge: number
4   }
5
6   const userOne: USER = {
7       |   userName: "Ari",
8       |   userAge: 21
9   }
10
```

# Union

Using Unions, we can define a variable that maybe of type 1 or type2 or so on. Pipe (|) operator is used to specify a union.

```
src > TS index.ts > ...
1   let nameOrAge : number | string;
2
3   nameOrAge = 21;
4   nameOrAge = "Ari";
5
6   Type 'boolean' is not assignable to type 'string | number'. ts(2322)
7   let nameOrAge: string | number
8
9   View Problem (⌘F8)  No quick fixes available
10  nameOrAge = true;
```

Let's combine this with what we learned.

```
src > TS index.ts > ...
1   type nameOrAge = number | string;
2   type obj = {
3     |   nameOrAge: nameOrAge
4   }
5
6   const user:obj = {
7     |   nameOrAge: 21
8   }
9
```

# Data Modifiers

In object-oriented programming, the concept of Encapsulation is used to make class members public or private i.e. a class can control the visibility of its data members. This is done using access modifiers. There are three types of access modifiers in TypeScript: public, private and protected.

```
src > TS index.ts > ...
1  class Employee {
2      // Public property: accessible from anywhere
3      public name: string;
4      // Private property: accessible only within this class
5      private readonly employeeId: number;
6      // Protected property: accessible within this class and subclasses
7      protected readonly position: string;
8      // Public method: accessible from anywhere
9      // SAME GOES FOR METHODS
10     constructor(name: string, employeeId: number, position: string) {
11         this.name = name;
12         this.employeeId = employeeId;
13         this.position = position;
14     }
15 }
```

We also have a modifier, “readonly”, which is used to make properties immutable after they are initialized.

```
src > TS index.ts > ...
1  class Person {
2      // Declaring a readonly property
3      public readonly id: number;
4      public readonly name: string;
5
6      constructor(id: number, name: string) {
7          this.id = id;
8          this.name = name;
9      }
10
11     // This method can read the readonly properties but cannot modify them
12     public displayInfo(): void {
13         console.log(`ID: ${this.id}, Name: ${this.name}`);
14     }
15 }
16
17 const person = new Person(1, 'Alice');
18 console.log(person.id); // Output: 1
19 console.log(person.name); // Output: Alice
20 person.id = 2; // Error: Cannot assign to 'id' because it is a read-only property
21
```



NANDRI