Learning TypeScript

Type-safe JavaScript

Josh Goldberg

Part I: Concepts

From JavaScript to TypeScript

Chapter 1

Vanilla JavaScript Pitfalls

- **Costly Freedom**

As the number of files grows in the project of JavaScript, you can only have vague ideas on how to call the functions.

```
function paintPainting(painter, painting) {
  return painter
    .prepare()
    .paint(painting, painter.ownMaterials)
    .finish();
}
```

You might even make a lucky guess that **painting** is a **string**.

Vanilla JavaScript Pitfalls

- **Loose Documentation**
  - There exists nothing in the JavaScript language specification to formalize description about code purpose.
  - Developers use JSDoc but it has key issues that often make it unpleasant to use in a large codebase
  - Maintaining JSDoc comments across a dozen files doesn’t take up too much time, but across hundreds or even thousands of constantly updating files can be a real chore.

Vanilla JavaScript Pitfalls

- **Weaker Developer Tooling**
  - Because JavaScript doesn’t provide built-in ways to identify types.
  - It can be difficult to automate large changes to or gain insights about a codebase.

TypeScript

- TypeScript was created internally at Microsoft in the early 2010s then released and open sourced in 2012.

- TypeScript is often described as a “superset of JavaScript” or “JavaScript with types.”

TypeScript

What is TypeScript

- Programming language - that includes all the existing JavaScript syntax, plus new TypeScript-specific syntax for defining and using types
- Type checker - It lets you know if it thinks anything is set up incorrectly
- Compiler - 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 such as VS Code how to provide helpful utilities to developers

Getting Started in the TypeScript Playground

The code is written in normal JavaScript syntax. If you tried to run that code in JavaScript, it would crash!

```
const firstName = "Georgia";
const nameLength = firstName.length();
//                           ~~~~~~
// This expression is not callable.
```

If you were to run the TypeScript type checker on this code, it would use its knowledge that the length property of a string is a number—not a function

Hovering over the code would give you the text of the complaint

```
const firstName = "Lizzo";
const nameLength = firstName.length();
```

```
(property) String.length: number
Returns the length of a String object.
This expression is not callable.
  Type 'Number' has no call signatures. ts(2349)
View Problem    No quick fixes available
```

Getting Started in the TypeScript Playground

Freedom Through Restriction

- TypeScript allows us to specify what types of values may be provided for parameters and variables.

- If you change the number of required parameters for a function, TypeScript will let you know if you forget to update a place that calls the function.

Getting Started in the TypeScript Playground

Freedom Through Restriction

- sayMyName was changed from taking in two parameters to taking one parameter, but the call to it with two strings wasn't updated and so is triggering a TypeScript complaint:
- That code would run without crashing in JavaScript, but its output would be different from expected (it wouldn't include "Knowles"):

```
// Previously: sayMyName(firstName, lastNameName) { ...
function sayMyName(fullName) {
  console.log(`You acting kind of shady, ain't callin' me ${fullName}`);
}

sayMyName("Beyoncé", "Knowles");
//                   ~~~~~~~~~
// Expected 1 argument, but got 2.
```

Getting Started in the TypeScript Playground

Precise Documentation

a TypeScript version of the paintPainting function from earlier.

```
interface Painter {
  finish(): boolean;
  ownMaterials: Material[];
  paint(painting: string, materials: Material[]): boolean;
}

function paintPainting(painter: Painter, painting: string): boolean { /* ...
*/ }
```

A TypeScript developer reading this code for the first time could understand that painter has at least three properties.

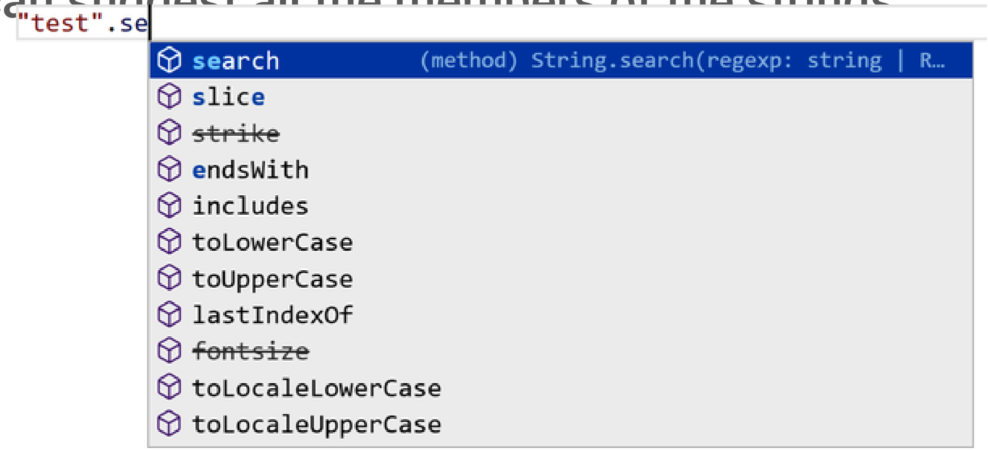
TypeScript provides an excellent, enforced system for describing how objects look.

Getting Started in the TypeScript Playground

Stronger Developer Tooling

TypeScript allow editors such as VS Code to gain much deeper insights into your code.

TypeScript can suggest all the members of the strings



Getting Started in the TypeScript Playground

Stronger Developer Tooling

When you add TypeScript's type checker for understanding code, it can give you these useful suggestions even for code you've written.

```
interface Painter {
  finish(): boolean;
  ownMaterials: Material[];
  paint(painting: string, materials: Material[]): boolean;
}

function paintPainting(painter: Painter, painting: string): boolean
  painter.
          finish              (method) Painter.finish(): boolean
          ownMaterials
          paint
```

Getting Started in the TypeScript Playground

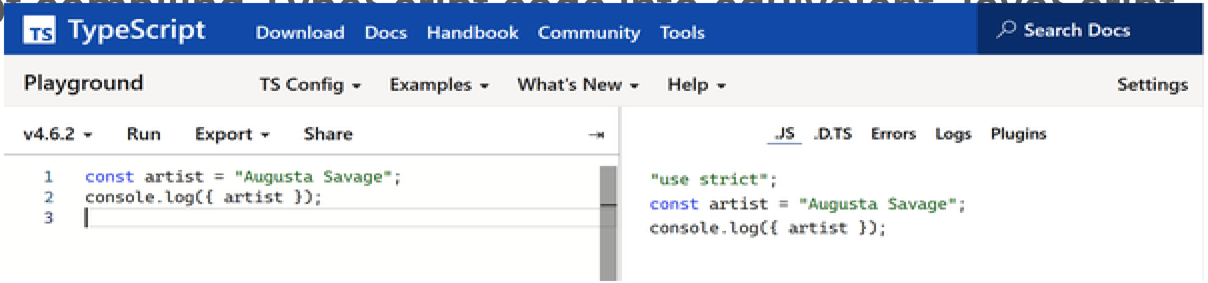
Compiling Syntax

TypeScript's compiler allows us to input TypeScript syntax, have it type checked, and get the equivalent JavaScript emitted.

```
const artist = "Augusta Savage";
console.log({ artist });
```

TypeScript Code

TypeScript compiling TypeScript code into equivalent JavaScript



Getting Started Locally

install the latest version of TypeScript globally

npm i -g typescript

run TypeScript on the command line with the tsc (TypeScript Compiler) command. Try it with the --version flag to make sure it's set up properly:

tsc --version

```
C:\>tsc --version
Version 4.8.2
```

Getting Started Locally

Running Locally

- Create a folder somewhere on your computer and run this command to create a new tsconfig.json configuration file:

  ```
  tsc --init
  ```

- A tsconfig.json file declares the settings that TypeScript uses when analyzing your code.
- Create a file named index.ts with the following contents:

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

- run tsc and provide it the name of that index.ts file:

  ```
  tsc index.ts
  ```

What TypeScript Is Not

Let’s discuss the limitations of TypeScript !

A Remedy for Bad Code

- TypeScript helps you structure your JavaScript, but other than enforcing type safety, it doesn’t enforce any opinions on what that structure should look like.

What TypeScript Is Not

Extensions to JavaScript (Mostly)

- TypeScript does not try to change how JavaScript works at all.
- TypeScript's design goals explicitly state that it should:
  - Align with current and future ECMAScript proposals
  - Preserve runtime behavior of all JavaScript code

What TypeScript Is Not

Slower Than JavaScript

- TypeScript is slow than JavaScript, That claim is generally inaccurate and misleading.

- The only changes TypeScript makes to code are if you ask it to compile your code down to earlier versions of JavaScript to support older runtime environments such as Internet Explorer 11.

- Browsers and Node.js, will run it.

What TypeScript Is Not

Finished Evolving

- The TypeScript language is constantly receiving bug fixes and feature additions to match the ever-shifting needs of the web community.

- The current version of the TypeScript is

```
C:\>tsc --version
Version 4.8.2
```

The Type System

Chapter 2

What’s in a Type?

- A “type” is a description of what a JavaScript value shape might be.
- “shape” means which properties and methods exist on a value.

- TypeScript understands the type of the value to be

  one of the seven basic primitives:

1. null; // null
2. undefined; // undefined
3. true; // boolean
4. "Louise"; // string
5. 1337; // number
6. 1337n; // bigint
7. Symbol("Franklin"); // symbol

What’s in a Type?

- If you hover your mouse over the variable’s name. The resultant popover will include the name of the primitive,

```
let singer: string
```

```
let singer = "Ella Fitzgerald";
```

- TypeScript knows that the ternary expression always results in a string, so the bestSong variable is a string:

```
let bestSong: string
```

```
let bestSong = Math.random() > 0.5
  ? "Chain of Fools"
  : "Respect";
```

What’s in a Type?

Type Systems

A type system is the set of rules for how a programming language understands what types the constructs in a program may have.

```
let firstName = "Whitney";
firstName.length();
//        ~~~~~~~
//  This expression is not callable.
//    Type 'Number' has no call signatures
```

TypeScript came to that complaint by, in order:

1. Reading in the code and understanding there to be a variable named firstName

2. Concluding that firstName is of type string because its initial value is a string, "Whitney"

3. Seeing that the code is trying to access a .length member of firstName and call it like a function

4. Complaining that the .length member of a string is a number, not a function (it can’t be called like a function)

What's in a Type?

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

```
let let wat;
//      ~~~
// Error: ',' expected.
```

```
console.blub("Nothing is worth more than laughter.");
//      ~~~~
// Error: Property 'blub' does not exist on type 'Console'.
```

Type

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

Assignability

TypeScript is fine with later assigning a different value of the same type to a

Variable.

If a variable is, say, initially a string value, later assigning it another string would be fine:

```
let firstName = "Carole";
firstName = "Joan";
```

If TypeScript sees an assignment of a different type, it will give us a type error.

```
let lastName = "King";
lastName = true;
// Error: Type 'boolean' is not assignable to type 'string'.
```

Assignability

Understanding Assignability Errors

when we wrote

lastName = true in the previous snippet,

we were trying to assign the value of true—type boolean—to the recipient variable lastName—type string.

Type Annotations

- Sometimes a variable doesn't have an initial value for TypeScript to read.
- It'll consider the variable by default to be implicitly the any type: indicating that it could be anything in the world.

```
let rocker; // Type: any

rocker = "Joan Jett"; // Type: string
rocker.toUpperCase(); // Ok

rocker = 19.58; // Type: number
rocker.toPrecision(1); // Ok

rocker.toUpperCase();
//     ~~~~~~~~~~~
// Error: 'toUpperCase' does not exist on type 'number'.
```

Type Annotations

- TypeScript provides a syntax for declaring the type of a variable without having to assign it an initial value, called a *type annotation*.
- A type annotation is placed after the name of a variable and includes a colon followed by the name of a type.

```
let rocker: string;
rocker = "Joan Jett";
```

- These type annotations exist only for TypeScript—they don't affect the runtime code and are not valid JavaScript syntax.

Type Annotations

Unnecessary Type Annotations

The following : string type annotation is redundant because TypeScript could already infer that firstName be of type string:

```
let firstName: string = "Tina";
//           ~~~~~~~~~ Does not change the type system...
```

Many developers generally prefer not to add type annotations on variables where the type annotations wouldn't change anything.

Type Annotations

Type Shapes

- TypeScript also knows what member properties should exist on objects.

- If you attempt to access a property of a variable, TypeScript will make sure that property is known to exist on that variable’s type.

Suppose we declare a rapper variable of type string. Later on, when we use that rapper variable, opera ows work on strings are allowed:

```
let rapper = "Queen Latifah";
rapper.length; // ok
```

Type Annotations

Modules

The JavaScript programming language did not include a specification for how files can share code between each other until relatively recently in its history.

Module

A file with a top-level export or import

Script

Any file that is not a module

Type Annotations

Modules

- Anything declared in a module file will be available only in that file unless an explicit export statement in that file exports it.
- A variable declared in one module with the same name as a variable declared in another file won't be considered a naming conflict (unless one file imports the other file's variable).

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

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

Type Annotations

Modules

- c.ts file causes a type error because it has a naming conflict between an imported shared and its own value:

```
// 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.
```

Type Annotations

Modules

- If a file is a script, all scripts have access to its contents.
- That means variables declared in a script file cannot have the same name as variables declared in other script files.

```
// a.ts
const shared = "Cher";
//    ~~~~~~
// Cannot redeclare block-scoped variable 'shared'.

// b.ts
const shared = "Cher";
//    ~~~~~~
// Cannot redeclare block-scoped variable 'shared'.
```

The a.ts and b.ts files are considered scripts because they do not have module-style export or import statements.

That means their variables of the same name conflict with each other as if they were declared in the same file:

Type Annotations

Modules

if you need a file to be a module without an export or import statement, you can add an export { }; somewhere in the file to force it to be a module:

```
// a.ts and b.ts
const shared = "Cher"; // Ok

export {};
```

Unions and Literals

Chapter 3

Union Types

- Take this mathematician variable:

```
let mathematician = Math.random() > 0.5
    ? undefined
    : "Mark Goldberg";
```

What type is mathematician?

mathematician can be either undefined or string. This kind of "either or" type is called a **union**.

- handle code cases where we don't know exactly which type a value is, but do know it's one of two or more options.
- TypeScript represents union types using the | (pipe) operator between the possible values, or constituents.

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

Union Types

Declaring Union Types

- Union types are an example of a situation when it might be useful to give an explicit type annotation for a variable even though it has an initial value.

```
let thinker: string | null = null;

if (Math.random() > 0.5) {
    thinker = "Susanne Langer"; // Ok
}
```

- thinker starts off null but is known to potentially contain a string instead.

Giving it an explicit string | null type annotation means TypeScript will allow it to be assigned values of type string:

Union Types

Union Properties

- TypeScript will only allow you to access member properties that exist on all possible types in the union.
- It will give you a type-checking error if you try to access a type that doesn't exist on all possible types.

Union Types

Union Properties

Example

```
let physicist = Math.random() > 0.5
    ? "Marie Curie"
    : 84;

physicist.toString(); // Ok

physicist.toUpperCase();
//        ~~~~~~~~~~~
// Error: Property 'toUpperCase' does not exist on type 'string | number'.
//   Property 'toUpperCase' does not exist on type 'number'.

physicist.toFixed();
//        ~~~~~~~
// Error: Property 'toFixed' does not exist on type 'string | number'.
//   Property 'toFixed' does not exist on type 'string'.
```

physicist is of type number | string. While .toString() exists in both types and is allowed to be used, (common properties)

.toUpperCase() and .toFixed() are not because .toUpperCase() is missing on the number type and .toFixed() is missing on the string type:

Narrowing

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

- A logical check that can be used to narrow types is called a type guard.

Narrowing

Assignment Narrowing

If you directly assign a value to a variable, TypeScript will narrow the variable's type to that value's type.

```
let admiral: number | string;

admiral = "Grace Hopper";

admiral.toUpperCase(); // Ok: string

admiral.toFixed();
//      ~~~~~~~~
// Error: Property 'toFixed' does not exist on type 'string'.
```

admiral variable is declared initially as a number | string, but after being assigned the value "Grace Hopper", TypeScript knows it must be a string:

Narrowing

Conditional Checks

if statement checking the variable for being equal to a known value.

```typescript
// Type of scientist: number | string
let scientist = Math.random() > 0.5
    ? "Rosalind Franklin"
    : 51;

if (scientist === "Rosalind Franklin") {
    // Type of scientist: string
    scientist.toUpperCase(); // Ok
}

// Type of scientist: number | string
scientist.toUpperCase();
//        ~~~~~~~~~~~
// Error: Property 'toUpperCase' does not exist on type 'string | number'.
//   Property 'toUpperCase' does not exist on type 'number'.
```

TypeScript is smart enough to understand that inside the body of that if statement, the variable must be the same type as the known value:

Narrowing

Typeof Checks

TypeScript also recognizes the typeof operator in narrowing down variable types.

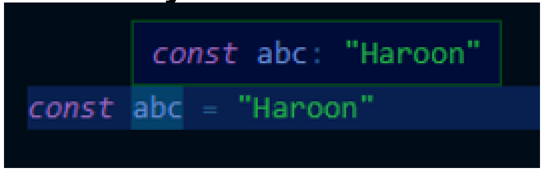
```
let researcher = Math.random() > 0.5
    ? "Rosalind Franklin"
    : 51;

if (typeof researcher === "string") {
    researcher.toUpperCase(); // Ok: string
}
```

checking if typeof researcher is "string" indicates to TypeScript that the type of researcher must be string:

Literal Types

- When you declare a variable via var you are telling the compiler that there is the chance that this variable will change its contents.
- In contrast, using const to declare a variable will inform TypeScript that this object will never change.
- A literal value type specifies a specific set of values and allows only those values.
- **Examples 1** → If you declare a variable as const and directly give it a literal value, TypeScript will infer the variable to be that literal value as a type.when you hover a mouse over a const variable with an initial literal Value, it will show you the variable’s type as that literal



Literal Types

- **Example 2** → TypeScript reporting a let variable as being generally its primitive type

```
let xyx: string
let xyx = "Hello"
```

- **Example 3** → a union of every possible matching literal value.

```
let abc: "Haroon" | "Abid" | "Majid"

abc = "Haroon"; //ok
abc = "Abid"; //ok
abc = "Majid"; //ok
abc = "Hamid"; // Not ok
```

```
Type '"Hamid"' is not assignable to type '"Haroon" | "Abid" | "Majid"'.
```

Translation: I was expecting a type matching A, but instead you passed B.

See full translation

```
let abc: "Haroon" | "Abid" | "Majid"
```

```
Type '"Hamid"' is not assignable to type '"Haroon" | "Abid" |
"Majid"'. ts(2322)
```

View Problem No quick fixes available

Literal Types

- **Example 4** → a union of literals and other data types (primitive types).

```
let abc: "Haroon" | number

abc = "Haroon"; //ok
abc = "Hamid"; // Not ok

let abc: number | "Haroon"
abc = 1234; //OK
```

Literal Types

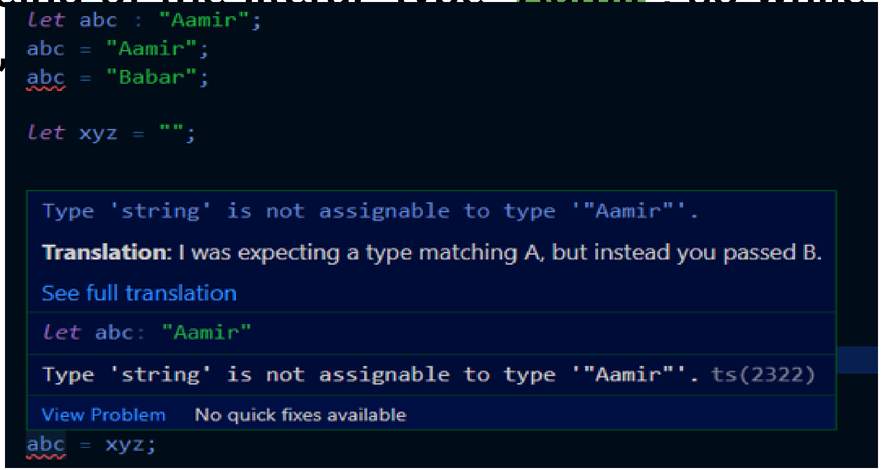
Literal Assignability

Different literal types within the same primitive type are not assignable to each other.

Example → Aamir is declared as being of the literal type "Aamir", so while the value "Aamir" may be given to it,

the types "Babar" and string are not

assignable to it:



Strict Null Checking

The Billion-Dollar Mistake

- The "billion-dollar mistake" is a industry term for many type systems allowing null values to be used in places that require a different type.

- In languages without strict null checking, code like this example that assign null to a string is allowed:

```
const firstName: string = null;
```

Strict Null Checking

The Billion-Dollar Mistake

- The "billion-dollar mistake" is a industry term for many type systems allowing null values to be used in places that require a different type.

- In languages without strict null checking, code like this example that assign null to a string is allowed:

```
const firstName: string = null;
```

Strict Null Checking

- **In strict null checking** mode, the null and undefined values are not in the domain of every type and are only assignable to themselves.
- The use of null and undefined can be restricted by enabling the strictNullChecks compiler setting (tsconfig.json)
- **Example** → with "strictNullChecks": false

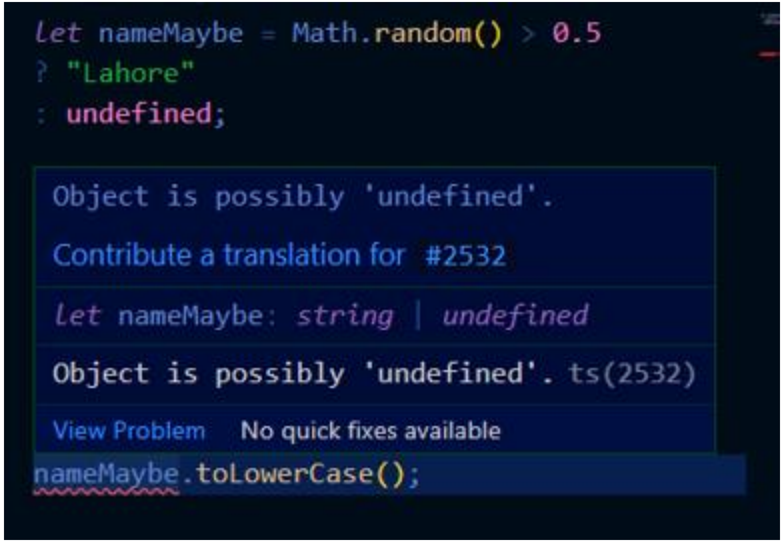
```
let nameMaybe = Math.random() > 0.5
? "Lahore"
: undefined;



nameMaybe.toLowerCase();
```

Strict Null Checking

- **Example → with "strictNullChecks": true**



Strict Null Checking

Truthiness Narrowing

- In this type of narrowing, we check whether a variable is truthy before using it.
- All values in JavaScript are truthy
- except for those defined as falsy: false, 0, 0n, "", null, undefined, and NaN

Strict Null Checking

Truthiness Narrowing

Example →

- geneticist is of type string | undefined
- undefined is always falsy

- TypeScript can deduce that it must be of type string within the if statement's body:

```
module01.ts > ...
let geneticist = Math.random() > 0.5
? "Barbara McClintock"
: undefined;
if (geneticist) {
geneticist.toUpperCase(); // Ok: string
}

Object is possibly 'undefined'.
Contribute a translation for #2532
let geneticist: string | undefined
Object is possibly 'undefined'. ts(2532)
View Problem    No quick fixes available
geneticist.toUpperCase();
```

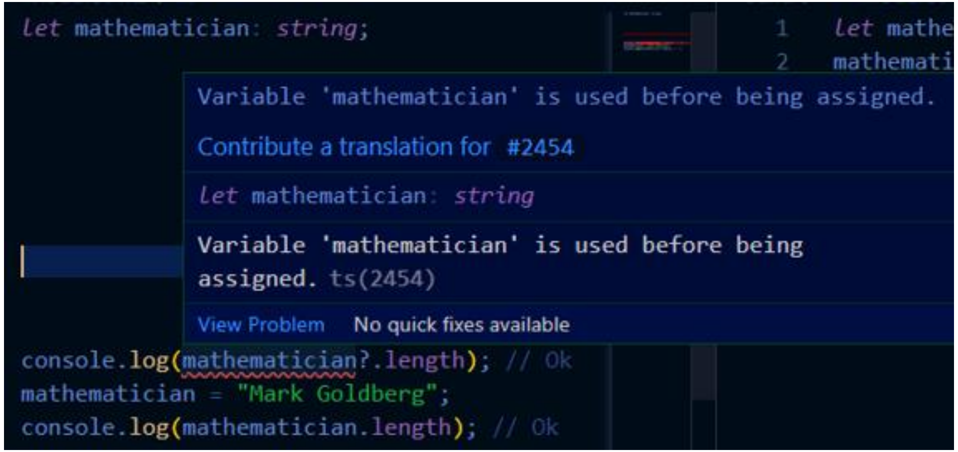
Strict Null Checking

Variables Without Initial Values

Declare its type but no value. In this case, the variable will be set to undefined.

var [identifier] : [type-annotation] ;

Example → TypeScript is smart enough to understand that the variable is undefined until a value is assigned. It will report a specialized error message if you try to use that variable



Type Aliases

- longer union types are inconvenient to type out repeatedly

Example 1→

```
let rawDataFirst: boolean | number | string | null | undefined;
let rawDataSecond: boolean | number | string | null | undefined;
let rawDataThird: boolean | number | string | null | undefined;
```

- A type alias starts with the type keyword, a new name, =, and then any type.

Example 2→

```
type RawData = boolean | number | string | null | undefined;

let rawDataFirst: RawData;
let rawDataSecond: RawData;
let rawDataThird: RawData;
```

Type Aliases

Example 3→

- Use type to declare flower as a type.
- By creating a type, you can use flower anywhere in your code, just like the primitive types (number, string, any etc)

```
type flower = "Rose" | "Tulip";

let flower1:flower="Rose"; //ok

    Type '"Lily"' is not assignable to type 'flower'.
    Translation: I was expecting a type matching A, but instead you passed B.
    See full translation
    let flower2: flower
    Type '"Lily"' is not assignable to type 'flower'. ts(2322)
    View Problem    No quick fixes available
let flower2:flower="Lily";
```

Type Aliases

Type Aliases Are Not JavaScript

- Type aliases, like type annotations, are not compiled to the output JavaScript.
- They exist purely in the TypeScript type system.

Type Aliases

Combining Type Aliases

Type aliases may reference other type aliases.

```
type Id = number | string;

// Equivalent to: number | string | undefined | null
type IdMaybe = Id | undefined | null;
```

This IdMaybe type is a union of the types within Id as well as undefined and null:

Objects

Chapter 4

Object Types

- In real life, a car is an object.
- A car has properties like weight and color, and methods like start and stop:

Object	Properties	Methods
	car.name = Fiat car.model = 500 car.weight = 850kg car.color = white	car.start() car.drive() car.brake() car.stop()

Object Types

- In real life, a car is an object.
- A car has properties like weight and color, and methods like start and stop:

Example 1 →

```
let employee: {pro1: string, pro2:number}

let var1= employee = {
    pro1:'77hgjghjg',
    pro2: 33


}
let var2= employee = {
    pro1: "new value",
    pro2: 32434
}
console.log(var1)
```

Object Types

Declaring Object Types

TypeScript can infer the types of properties based on their values.

Example 2 →

```
const fruits = {
    fruit1: "Apple",
    fruit2: "banana"
}
fruits.fruit1 = "orange" //OK
fruits.fruit1 = 133;    //Type 'number' is not assignable to type 'string'.
```

Object Types

Example 3 → poet variable is the same type with name: string and BirthYear: number:

```
let poet: {BirthY: number, name: string};
poet = {BirthY: 1797, name: "mirza Ghalib"}
poet = "Iqbal"  //Type 'string' is not assignable to type '
                //{ BirthY: number; name: string; }'.
```

Object Types

Aliased Object Types

We can avoid repeated typing properties of object with the help of **Aliases.**

```
//Alliased object type
type poet={BirthY: number, name: string};
let newPoet: poet;
newPoet = {BirthY: 1950, name: "name of poet"}
```

Structural Typing

- TypeScript’s type system is structurally typed.

- In structurally-typed languages, values are considered to be of equivalent types if all of their component features are of the same type.

- It’s mean when you declare a parameter or variable is of a particular object type, you’re telling TypeScript that whatever objects you use, they need to have those properties.

Structural Typing

Example 1→

In this TypeScript example you can see that a variable declared as the person type is assignable to a variable of the employee type

```
type person ={
    name: string,
    DOB: number
}
type employee = {
    name: string,
    DOB: number,
    new_employee: boolean
}
const var1: employee = {
    name:"",
    DOB: 33,
    new_employee: true

}
let var3 = var1;        // var1---> employee alias
var3 ={                 //new_employee is missing
    name: "",
    DOB: 212
}
console.log(var3.DOB)
```

Structural Typing

Duck Typing vs Structural Typing vs Nominal Typing

- Programming languages can be classified as **duck typed**, **structural typed**, or **nominal typed**.

Duck Typing

- Duck Typed languages use the Duck Test to evaluate whether the object can be evaluated as a particular type. Duck Test states:

If it looks like a duck, swims like a duck,
and quacks like a duck, then it probably is a duck.

Duck-Typed languages provide the most flexibility to the programmer. And the programmers need to write the least amount of code. But these languages can be unsafe and can create runtime errors.

Structural Typing

Duck Typing vs Structural Typing vs Nominal Typing

Nominal Typing

- Nominal-Typed languages mandate programmers to explicitly call the type — but it means more code and less flexibility (additional dependencies).

Structural Typing

- Structural-Typed languages provide a balance — it has required compile-time checks and doesn't require explicit declaration of the dependencies.

In summary: JavaScript is duck typed whereas TypeScript is structurally typed.

Structural Typing

Usage Checking

TypeScript will check that the value is assignable to that object type.

The value must have the required properties of object type.

If any member required on the object type is missing in the object, TypeScript will issue a type error.

Structural Typing

Usage Checking

Example →

```
type FirstAndLastNames = {
    frist: string,
    second:string
}
const hasBoth: FirstAndLastNames ={ //ok USAGE
    frist: "name1",
    second: "name2"
}
const hasOnlyOne: FirstAndLastNames ={ //Property 'second' is missing in type
    frist: ""
}
```

Structural Typing

Excess Property Checking

Typescript will report a type error if a variable is declared with an object type and its initial value has more fields than its type describes.

Structural Typing

Excess Property Checking

Example →

```
type FirstAndLastNames = {
    frist: string,
    second:string
}
const hasBoth: FirstAndLastNames ={ //ok USAGE
    frist: "name1",
    second: "name2"
}
const hasOnlyOne: FirstAndLastNames ={
    frist: "new name",
    second: "ew 2nd name",
    third: ""   //Type '{ frist: string; second: string; third: string; }'
                //is not assignable to type 'FirstAndLastNames'.
}
```

Structural Typing

Nested Object Types

TypeScript’s object types must be able to represent nested object types in the type system.

Example→

```
type Poem = {
    author: {firstName: string, lastName: string} //nested object
    poetry_name: string
}
const poemMatch: Poem = {            ///OK as per signature
    author: {firstName: "1st Name", lastName: "2nd Name"},
    poetry_name: "poetry name"
}
const poemMismatch: Poem = {
    author: {firstName: "1st Name"}, //Property 'lastName' is missing in type
}
```

Structural Typing

Optional Properties

- Object type properties don’t all have to be required in the object.
- You can include a ? before the : in a type property’s type annotation to indicate that it’s an optional property.

Structural Typing

Optional Properties

Example 1 → Book type requires only a pages property and optionally allows an author. Objects adhering to it may provide author or leave it out as long as they provide pages:

```
type Book = {
    author?: string
    pages: number
}
const book1: Book={              //ok as per definition
    author: "author name",
    pages: 233
}
const book2: Book ={             //still OK, even author is no mentioned
    pages: 333                   //author property is optional
}
```

Unions of Object Types

- In TypeScript code you can describe a type that can be one or more different object types that have slightly different properties.

Unions of Object Types

Inferred Object-Type Unions

If a variable is given an initial value that could be one of multiple object types, TypeScript will infer its type to be a union of object types.

Unions of Object Types

Inferred Object-Type Unions

Example → poem value always has a name property of type string, and may or may not have pages and rhymes properties:

```
//Unions of Object Types
//Inferred Object Types Unions
const poem = Math.random() >.05
? {name: "name one", pages: 234}
: {name: "name second", rhymes: true}
            // const poem: {
            //     name: string;
            //     pages: number;
            //     rhymes?: undefined;
            // } | {
            //     name: string;
            //     rhymes: boolean;
            //     pages?: undefined;
            // }
console.log(typeof poem.name)
console.log(typeof poem.pages)
console.log(typeof poem.rhymes)
```

Unions of Object Types

Explicit Object-Type Unions

Example →

poem variable is explicitly typed to be a union type that always has property along with either pages or rhymes. Accessing names is allowed because it always exists, but pages and rhymes aren't guaranteed to exist:

```
type PoemWithPages ={
    name: string,
    pages: number
}
type PoemWithRhymes ={
    name: string,
    rhymes: boolean
}

type Poem = PoemWithPages | PoemWithRhymes;
const var1: Poem = Math.random()>0.5
?{name: "name one", pages: 778}
:{name: "second name", rhymes:true}

var1.name;
var1.pages; //Property 'pages' does not exist on type 'Poem'.
```

Unions of Object Types

Narrowing Object Types

If the type checker sees that an area of code can only be run

if a union typed value contains a certain property, it will narrow the value's type to only the constituents that contain

```
type PoemWithPages ={
        name: string,
        pages: number
    }
    type PoemWithRhymes ={
        name: string,
        rhymes: boolean
    }

    type Poem = PoemWithPages | PoemWithRhymes;
    const var1: Poem = Math.random()>0.5
    ?{name: "name one", pages: 778}
    :{name: "second name", rhymes:true}

    if ("pages" in var1){
        var1.pages  ///OK: var1 is narrowed to PoemWithPages
    }else{
        var1.rhymes  ///OK: var1 is narrowed to PoemWithRhymes
    }
```

Unions of Object Types

Discriminated Unions

- Literal types which you can use to let TypeScript narrow down the possible current type. This kind of type is called a discriminated union.
- the property whose value indicates the object’s type is a **discriminant**.

Unions of Object Types

Discriminated Unions

Example →

```
type LowRain = {
    flood: string,
    location: string
}
type HighRain = {
    flood: string,
    rain_mm: number
}
type Rain = LowRain | HighRain;
const var1: Rain ={
    flood: "Heavy Rain", location: "Sindh", rain_mm: 100
}
var1.flood      //OK
var1.location   // because of discriminated union
                //Property 'location' does not exist on type 'Rain'.
```

Intersection Types

- TypeScript allows representing a type that is multiple types at the same time: an & intersection type.
- Intersection types are typically used with **aliased** object types to create a new type that combines multiple existing object types.

Intersection Types

Example →

```
type ArtWork= {
    pro1: string,
    pro2: string
};
type Writing = {
    pro3: number,
    pro2: string
}

type newType = ArtWork & Writing;
const var1: newType={pro1: "",pro2: "", pro3:23}
var1.pro1;           //ok
var1.pro2;           //ok
var1.pro3;           //ok
```

Intersection Types

Dangers of Intersection Types

1. Long assignability errors

```
type ShortPoemBase = { author: string };
type Haiku = ShortPoemBase & { kigo: string; type: "haiku" };
type Villanelle = ShortPoemBase & { meter: number; type: "villanelle" };
type ShortPoem = Haiku | Villanelle;

const oneArt: ShortPoem = {
    author: "Elizabeth Bishop",
    type: "villanelle",
};
// Type '{ author: string; type: "villanelle"; }'
// is not assignable to type 'ShortPoem'.
//   Type '{ author: string; type: "villanelle"; }'
//   is not assignable to type 'Villanelle'.
//     Property 'meter' is missing in type
//     '{ author: string; type: "villanelle"; }'
//     but required in type '{ meter: number; type: "villanelle"; }'.
```

Intersection Types

Dangers of Intersection Types

2. Never

Trying to & two primitive types together will result in the never type, represented by the keyword never:

Example →

```
type NotPossible = number & string;
// Type: never
```

Part II: Features

Functions

Chapter 5

Function Parameters

JavaScript Func

Example →

```
function sing(song) {
  console.log(`Singing: ${song}!`);
}
```

<u>**Problem in code:**</u>

- **Without explicit type information declared, we may never know—**
- **TypeScript will consider it to be the any type, meaning the parameter's type could be anything.**

Function Parameters

TypeScript Function

```
function sing(song: string) {
    console.log(`Singing: ${song}!`);
}
```

Example →

<u>Solution of previous code:</u>

- TypeScript allows you to declare the type of function parameters with a type annotation.
- we can use a **:** string to tell TypeScript that the song parameter is of type **string**.

Function Parameters

Required Parameters

TypeScript's argument counting will come into play if a function is called with either too few or too many arguments.

Example →

```
function singtwo(first: string, second: string){
    console.log(first, second);
    }
singtwo("dfd","sder")
//singtwo("3sdfs") ///Expected 2 arguments, but got 1.
singtwo(222,232)    //error TS2345: Argument of type 'number' is not assignable
                    //to parameter of type 'string'
```

Function Parameters

Optional Parameters

TypeScript allows annotating a parameter as optional by adding a ? before the : in its type annotation

Example →

```
function announceSong(song: string, singer?: string) {
    console.log(`Song: ${song}`);
    if (singer) {
    console.log(`Singer: ${singer}`);
    }
    }
    announceSong("Greensleeves"); // Ok
    announceSong("Greensleeves", undefined); // Ok
    announceSong("Chandelier", "Sia"); // Ok
```

Function Parameters

Default Parameters

- TypeScript may be given a default value with an = and a value in their declaration.
- TypeScript will infer the parameter’s type based on that default value.

Example →

```
function rateSong(song:string, rating = 0){
    console.log(`${song} gets ${rating}`)
}
rateSong("song name") //ok
rateSong("song name", 34343) //ok
rateSong("'dfdfd", undefined) //ok
rateSong("song name", "3df") //error TS2345: Argument of type
                              //'number' is not assignable
//                            //to parameter of type 'string'
```

Function Parameters

Rest Parameters

- Some functions are made to be called with any number of arguments.
- The `...` spread operator may be placed on the last parameter in a function declaration to indicate any “rest” arguments, with a `[ ]` syntax added at the end to indicate it’s an array of arguments.

Function Parameters

Rest Parameters

Example →

```
function singAllTheSongs(singer: string, ...songs: string[]) {
    for (const song of songs) {
    console.log(`${song}, by ${singer}`);
    }
    }
    singAllTheSongs("Strings"); // Ok
    singAllTheSongs("Shehzad Roy", "Laga Reh", "Humari Shaan", ); // Ok
    singAllTheSongs("Vital Sign", 2000);
    // ~~~~~
    // Error: Argument of type 'number' is not
    // assignable to parameter of type 'string'.
```

```
singSong (["kd",'', ''],'', 1223)   //error TS2554: Expected 1 arguments,
                                     //but got 3.
```

Return Types

- If TypeScript understands all the possible values returned by a function, it'll know what type the function returns.

Example →singSongs is understood by TypeScript to return a number:

```
// Type: (songs: string[]) => number
function singSongs(songs: string[]) {
  for (const song of songs) {
    console.log(`${song}`);
  }

  return songs.length;
}
```

- If a function contains multiple return statements with different values, TypeScript will infer the return type to be a union of all the possible returned types.

Return Types

Explicit Return Types

- This ensures that the return value is assigned to a variable of the correct type; or in the ere is no return value,

Exa

❌ Incorrect

```
// Should indicate that no value is returned (void)
function test() {
  return;
}

// Should indicate that a number is returned
var fn = function () {
  return 1;
};
```

✅ Correct

```
// No return value should be expected (void)
function test(): void {
  return;
}

// A return value of type number
var fn = function (): number {
  return 1;
};
```

Return Types

Explicit Return Types

- This ensures that the return value is assigned to a variable of the correct type; or in the ere is no return value,

❌ Incorrect

✅ Correct

Example →

‘

```
// Should indicate that no value is returned (void)
function test() {
  return;
}

// Should indicate that a number is returned
var fn = function () {
  return 1;
};
```

```
// No return value should be expected (void)
function test(): void {
  return;
}

// A return value of type number
var fn = function (): number {
  return 1;
};
```

Return Types

Explicit Return Types

Example 2 →Here, the getSongRecordingDate function is explicitly declared as returning **Date | undefined**, but one of its return statements incorrectly provides a **string**:

```
function getSongRecordingDate(song: string): Date | undefined {
  switch (song) {
    case "Strange Fruit":
      return new Date('April 20, 1939'); // Ok

    case "Greensleeves":
      return "unknown";
      // Error: Type 'string' is not assignable to type 'Date'.

    default:
      return undefined; // Ok
  }
}
```

‘

Function Types

- Function type syntax looks similar to an arrow function, but with a type instead of the body.

Example→ nothingInGivesString variable's type describes a function with no parameters and a returned **string** value:

```
let nothingInGivesString: () => string;
```

'

Function Types

callback parameters

A callback function is defined as a function passed into another function as an argument, which is then invoked inside the outer function to complete the desirable routine or action.

```
function outerFunction(callback: () => void) {
  callback();
}
```

‘

Function Types

Function Type Parentheses

- Function types may be placed anywhere that another type would be used. That includes union types.
- In union types, parentheses may be used to indicate which part of an annotation is the function return or the surrounding union type:

Example →

```
// Type is a function that returns a union: string | undefined
let returnsStringOrUndefined: () => string | undefined;

// Type is either undefined or a function that returns a string
let maybeReturnsString: (() => string) | undefined;
```

Function Types

Parameter Type Inferences

- TypeScript can infer the types of parameters in a function

Example → the **song** and **index** parameters here are inferred by TypeScript to be **string** and **number**, respectively:

```
const songs = ["Call Me", "Jolene", "The Chain"];
// song: string
// index: number
songs.forEach((song, index) => {
console.log(`${song} is at index ${index}`);
});
```

Function Types

Function Type Aliases

- Type aliases can be used for function types as well.

Example → function parameters can themselves be typed with aliases that happen to refer to a function type. This usesNumberToString function has a single parameter which is itself the NumberToString aliased function type:

```
type NumberToString = (input: number) => string;
function usesNumberToString(numberToString: NumberToString) {
console.log(`The string is: ${numberToString(1234)}`);
}
usesNumberToString((input) => `${input}! Number arrived!`); // Ok
usesNumberToString((input) => input * 2);
// ~~~~~~~~~~
// Error: Type 'number' is not assignable to type 'string'.
```

More Return Types

Void Returns

- Some functions aren't meant to return any value.
- 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 such a function that returns nothing.
- The void type is not JavaScript. It's a TypeScript keyword used to declare return types of functions.

More Return Types

Void Returns

- void indicates that any returned value from the function would be ignored.

Example 1→ songLogger variable represents a function that takes in a song: string and doesn't return a value:

```
let songLogger: (song: string) => void;
songLogger = (song) => {
console.log(`${songs}`);
};
songLogger("Heart of Glass"); // Ok
```

More Return Types

Void Returns

Example 2 → Trying to assign a value of type **void** to a value whose type instead includes **undefined** is a type error:

```
function returnsVoid() {
    return;
    }
    let lazyValue: string | undefined;
    lazyValue = returnsVoid();
    // Error: Type 'void' is not assignable to type 'string | undefined'.
```

More Return Types

Never Returns

- TypeScript introduced a new type **never**, which indicates the values that will never occur.
- The **never** type contains no value.
- The **never** type represents the return type of a function that always throws an error or a function that contains an indefinite loop.

Example 1 → Typically, you use the never type to represent the return type of a function that always throws an error.

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

Function Overloads

- Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters.
- When a function name is overloaded with different jobs it is called Function Overloading.

Example 1→

```
function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {
    return a + b;
}

console.log(add("Hello ", "Steve")); // returns "Hello Steve"
console.log(add(10, 20)); // returns 30
```

Function Overloads

Example 2→ Function overloading with different number of parameters and types with same name is not supported.

```
function display(a:string, b:string):void //Compiler Error: Duplicate function implementation
{
    console.log(a + b);
}
function display(a:number): void //Compiler Error: Duplicate function implementation
{
    console.log(a);
}
```

Function Overloads

WARNING

Function overloads are generally used as a last resort for complex, difficult-to-describe function types. It's generally better to keep functions simple and avoid using function overloads when possible.