The Concise TypeScript Book

Simone Poggiali

Table of Contents

About This Book

This book has been created from gibbok/typescript-book as a demonstration of Typesetter.

Thanks to Simone Poggiali for all his hard work!

~ Aaron Saray

Introduction

Welcome to The Concise TypeScript Book! This guide equips you with essential knowledge and practical skills for effective TypeScript development. Discover key concepts and techniques to write clean, robust code. Whether you're a beginner or an experienced developer, this book serves as both a comprehensive guide and a handy reference for leveraging TypeScript's power in your projects.

This book covers TypeScript 5.2.

About the Author

Simone Poggiali is an experienced Senior Front-end Developer with a passion for writing professional-grade code since the 90s. Throughout his international career, he has contributed to numerous projects for a wide range of clients, from startups to large organizations. Notable companies such as HelloFresh, Siemens, O2, and Leroy Merlin have benefited from his expertise and dedication.

You can reach Simone Poggiali on the following platforms:

- LinkedIn: https://www.linkedin.com/in/simone-poggiali
- GitHub: https://github.com/gibbok
- Twitter: https://twitter.com/gibbok_coding
- Email: gibbok.coding□gmail.com

TypeScript Introduction

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript. It was originally designed by Anders Hejlsberg in 2012 and is currently developed and maintained by Microsoft as an open source project.

TypeScript compiles to JavaScript and can be executed in any JavaScript runtime (e.g., a browser or server Node.js).

TypeScript supports multiple programming paradigms such as functional, generic, imperative, and object-oriented. TypeScript is neither an interpreted nor a compiled language.

Why TypeScript?

TypeScript is a strongly typed language that helps prevent common programming mistakes and avoid certain kinds of run-time errors before the program is executed.

A strongly typed language allows the developer to specify various program constraints and behaviors in the data type definitions, facilitating the ability to verify the correctness of the software and prevent defects. This is especially valuable in large-scale applications.

Some of the benefits of TypeScript:

- Static typing, optionally strongly typed
- Type Inference
- Access to ES6 and ES7 features
- Cross-Platform and Cross-browser Compatibility
- Tooling support with IntelliSense

TypeScript and JavaScript

TypeScript is written in `.ts` or `.tsx` files, while JavaScript files are written in `.js` or `.jsx`.

Files with the extension `.tsx` or `.jsx` can contain JavaScript Syntax Extension JSX, which is used in React for UI development.

TypeScript is a typed superset of JavaScript (ECMAScript 2015) in terms of syntax. All JavaScript code is valid TypeScript code, but the reverse is not always true.

For instance, consider a function in a JavaScript file with the `.js` extension, such as the following:

```
const sum = (a, b) => a + b;
```

The function can be converted and used in TypeScript by changing the file extension to `.ts`. However, if the same function is annotated with TypeScript types, it cannot be executed in any JavaScript runtime without compilation. The following TypeScript code will produce a syntax error if it is not compiled:

```
const sum = (a: number, b: number): number => a + b;
```

TypeScript was designed to detect possible exceptions that can occur at runtime during compilation time by having the developer define the intent with type annotations. In addition, TypeScript can also catch issues if no type annotation is provided. For instance, the following code snippet does not specify any TypeScript types:

```
const items = [{ x: 1 }, { x: 2 }];
const result = items.filter(item => item.y);
```

In this case, TypeScript detects an error and reports:

```
Property 'y' does not exist on type '{ x: number; }'.
```

TypeScript's type system is largely influenced by the runtime behavior of JavaScript. For example, the addition operator (+), which in JavaScript can either perform string concatenation or numeric addition, is modeled in the same way in TypeScript:

```
const result = '1' + 1; // Result is of type string
```

The team behind TypeScript has made a deliberate decision to flag unusual usage of JavaScript as errors. For instance, consider the following valid JavaScript code:

```
const result = 1 + true; // In JavaScript, the result is equal 2
```

However, TypeScript throws an error:

```
Operator '+' cannot be applied to types 'number' and 'boolean'.
```

This error occurs because TypeScript strictly enforces type compatibility, and in this case, it identifies an invalid operation between a number and a boolean.

TypeScript Code Generation

The TypeScript compiler has two main responsibilities: checking for type errors and compiling to JavaScript. These two processes are independent of each other. Types do not affect the execution of the code in a JavaScript runtime, as they are completely erased

during compilation. TypeScript can still output JavaScript even in the presence of type errors. Here is an example of TypeScript code with a type error:

```
const add = (a: number, b: number): number => a + b;
const result = add('x', 'y'); // Argument of type 'string' is not
assignable to parameter of type 'number'.
```

However, it can still produce executable JavaScript output:

```
'use strict';
const add = (a, b) => a + b;
const result = add('x', 'y'); // xy
```

It is not possible to check TypeScript types at runtime. For example:

```
interface Animal {
    name: string;
}
interface Dog extends Animal {
    bark: () => void;
}
interface Cat extends Animal {
    meow: () => void;
}
const makeNoise = (animal: Animal) => {
    if (animal instanceof Dog) {
        // 'Dog' only refers to a type, but is being used as a value here.
        // ...
    }
};
```

As the types are erased after compilation, there is no way to run this code in JavaScript. To recognize types at runtime, we need to use another mechanism. TypeScript provides several options, with a common one being "tagged union". For example:

```
interface Dog {
    kind: 'dog'; // Tagged union
    bark: () => void;
}
interface Cat {
    kind: 'cat'; // Tagged union
    meow: () => void;
}
type Animal = Dog | Cat;

const makeNoise = (animal: Animal) => {
    if (animal.kind === 'dog') {
        animal.bark();
    } else {
```

```
        animal.meow();
    }
};

const dog: Dog = {
    kind: 'dog',
    bark: () => console.log('bark'),
};
makeNoise(dog);
```

The property "kind" is a value that can be used at runtime to distinguish between objects in JavaScript.

It is also possible for a value at runtime to have a type different from the one declared in the type declaration. For instance, if the developer has misinterpreted an API type and annotated it incorrectly.

TypeScript is a superset of JavaScript, so the "class" keyword can be used as a type and value at runtime.

```
class Animal {
    constructor(public name: string) {}
}
class Dog extends Animal {
    constructor(
        public name: string,
        public bark: () => void
    ) {
        super(name);
    }
}
class Cat extends Animal {
    constructor(
        public name: string,
        public meow: () => void
    ) {
        super(name);
    }
}
type Mammal = Dog | Cat;

const makeNoise = (mammal: Mammal) => {
    if (mammal instanceof Dog) {
        mammal.bark();
    } else {
        mammal.meow();
    }
};

const dog = new Dog('Fido', () => console.log('bark'));
```

```
makeNoise(dog);
```

In JavaScript, a "class" has a "prototype" property, and the "instanceof" operator can be used to test if the prototype property of a constructor appears anywhere in the prototype chain of an object.

TypeScript has no effect on runtime performance, as all types will be erased. However, TypeScript does introduce some build time overhead.

Modern JavaScript Now (Downleveling)

TypeScript can compile code to any released version of JavaScript since ECMAScript 3 (1999). This means that TypeScript can transpile code from the latest JavaScript features to older versions, a process known as Downleveling. This allows the usage of modern JavaScript while maintaining maximum compatibility with older runtime environments.

It's important to note that during transpilation to an older version of JavaScript, TypeScript may generate code that could incur a performance overhead compared to native implementations.

Here are some of the modern JavaScript features that can be used in TypeScript:

- ECMAScript modules instead of AMD-style "define" callbacks or CommonJS "require" statements.
- Classes instead of prototypes.
- Variables declaration using "let" or "const" instead of "var".
- "for-of" loop or ".forEach" instead of the traditional "for" loop.
- Arrow functions instead of function expressions.
- Destructuring assignment.
- Shorthand property/method names and computed property names.
- Default function parameters.

By leveraging these modern JavaScript features, developers can write more expressive and concise code in TypeScript.

Getting Started With TypeScript

Installation

Visual Studio Code provides excellent support for the TypeScript language but does not include the TypeScript compiler. To install the TypeScript compiler, you can use a package manager like npm or yarn:

```
npm install typescript --save-dev
```

or

```
yarn add typescript --dev
```

Make sure to commit the generated lockfile to ensure that every team member uses the same version of TypeScript.

To run the TypeScript compiler, you can use the following commands

```
npx tsc
```

or

```
yarn tsc
```

It is recommended to install TypeScript project-wise rather than globally, as it provides a more predictable build process. However, for one-off occasions, you can use the following command:

```
npx tsc
```

or installing it globally:

```
npm install -g typescript
```

If you are using Microsoft Visual Studio, you can obtain TypeScript as a package in NuGet for your MSBuild projects. In the NuGet Package Manager Console, run the following command:

```
Install-Package Microsoft.TypeScript.MSBuild
```

During the TypeScript installation, two executables are installed: "tsc" as the TypeScript compiler and "tsserver" as the TypeScript standalone server. The standalone server contains the compiler and language services that can be utilized by editors and IDEs to provide intelligent code completion.

Additionally, there are several TypeScript-compatible transpilers available, such as Babel

(via a plugin) or swc. These transpilers can be used to convert TypeScript code into other target languages or versions.

Configuration

TypeScript can be configured using the tsc CLI options or by utilizing a dedicated configuration file called tsconfig.json placed in the root of the project.

To generate a tsconfig.json file prepopulated with recommended settings, you can use the following command:

```
tsc --init
```

When executing the `tsc` command locally, TypeScript will compile the code using the configuration specified in the nearest tsconfig.json file.

Here are some examples of CLI commands that run with the default settings:

```
tsc main.ts // Compile a specific file (main.ts) to JavaScript
tsc src/*.ts // Compile any .ts files under the 'src' folder to JavaScript
tsc app.ts util.ts --outfile index.js // Compile two TypeScript files
(app.ts and util.ts) into a single JavaScript file (index.js)
```

TypeScript Configuration File

A tsconfig.json file is used to configure the TypeScript Compiler (tsc). Usually, it is added to the root of the project, together with the `package.json` file.

Notes:

- tsconfig.json accepts comments even if it is in json format.
- It is advisable to use this configuration file instead of the command-line options.

At the following link you can find the complete documentation and its schema:

https://www.typescriptlang.org/tsconfig

http://json.schemastore.org/tsconfig

The following represents a list of the common and useful configurations:

target

The "target" property is used to specify which version of JavaScript ECMAScript version your TypeScript should emit/compile into. For modern browsers ES6 is a good option, for older browsers, ES5 is recommended.

lib

The "lib" property is used to specify which library files to include at compilation time. TypeScript automatically includes APIs for features specified in the "target" property, but it is possible to omit or pick specific libraries for particular needs. For instance, if you are working on a server project, you could exclude the "DOM" library, which is useful only in a browser environment.

strict

The "strict" property enables stronger guarantees and enhances type safety. It is advisable to always include this property in your project's tsconfig.json file. Enabling the "strict" property allows TypeScript to:

- Emit code using "use strict" for each source file.
- Consider "null" and "undefined" in the type checking process.
- Disable the usage of the "any" type when no type annotations are present.
- Raise an error on the usage of the "this" expression, which would otherwise imply the "any" type.

module

The "module" property sets the module system supported for the compiled program. During runtime, a module loader is used to locate and execute dependencies based on the specified module system.

The most common module loaders used in JavaScript are Node.js CommonJS for server-side applications and RequireJS for AMD modules in browser-based web applications. TypeScript can emit code for various module systems, including UMD, System, ESNext, ES2015/ES6, and ES2020.

Note: The module system should be chosen based on the target environment and the module loading mechanism available in that environment.

moduleResolution

The "moduleResolution" property specifies the module resolution strategy. Use "node" for modern TypeScript code, the "classic" strategy is used only for old versions of TypeScript (before 1.6).

esModuleInterop

The "esModuleInterop" property allows import default from CommonJS modules that did not export using the "default" property, this property provides a shim to ensure compatibility in the emitted JavaScript. After enabling this option we can use `import MyLibrary from "my-library"` instead of `import * as MyLibrary from "my-library"`.

jsx

The "jsx" property applies only to .tsx files used in ReactJS and controls how JSX constructs are compiled into JavaScript. A common option is "preserve" which will compile to a .jsx file keeping unchanged the JSX so it can be passed to different tools like Babel for further transformations.

skipLibCheck

The "skipLibCheck'' property will prevent TypeScript from type-checking the entire imported third-party packages. This property will reduce the compile time of a project. TypeScript will still check your code against the type definitions provided by these packages.

files

The "files" property indicates to the compiler a list of files that must always be included in the program.

include

The "include" property indicates to the compiler a list of files that we would like to include. This property allows glob-like patterns, such as "*" *for any subdirectory,* "" for any file name, and "?" for optional characters.

exclude

The "exclude" property indicates to the compiler a list of files that should not be included in the compilation. This can include files such as "node_modules" or test files. Note: tsconfig.json allows comments.

importHelpers

TypeScript uses helper code when generating code for certain advanced or down-leveled JavaScript features. By default, these helpers are duplicated in files using them. The `importHelpers` option imports these helpers from the `tslib` module instead, making the JavaScript output more efficient.

Migration to TypeScript Advice

For large projects, it is recommended to adopt a gradual transition where TypeScript and JavaScript code will initially coexist. Only small projects can be migrated to TypeScript in one go.

The first step of this transition is to introduce TypeScript into the build chain process. This can be done by using the "allowJs" compiler option, which permits .ts and .tsx files to coexist with existing JavaScript files. As TypeScript will fall back to a type of "any" for a variable when it cannot infer the type from JavaScript files, it is recommended to disable

"noImplicitAny" in your compiler options at the beginning of the migration.

The second step is to ensure that your JavaScript tests work alongside TypeScript files so that you can run tests as you convert each module. If you are using Jest, consider using `ts-jest`, which allows you to test TypeScript projects with Jest.

The third step is to include type declarations for third-party libraries in your project. These declarations can be found either bundled or on DefinitelyTyped. You can search for them using https://www.typescriptlang.org/dt/search and install them using:

```
npm install --save-dev @types/package-name or yarn add --dev
@types/package-name.
```

The fourth step is to migrate module by module with a bottom-up approach, following your Dependency Graph starting with the leaves. The idea is to start converting Modules that do not depend on other Modules. To visualize the dependency graphs, you can use the "madge" tool.

Good candidate modules for these initial conversions are utility functions and code related to external APIs or specifications. It is possible to automatically generate TypeScript type definitions from Swagger contracts, GraphQL or JSON schemas to be included in your project.

When there are no specifications or official schemas available, you can generate types from raw data, such as JSON returned by a server. However, it is recommended to generate types from specifications instead of data to avoid missing edge cases.

During the migration, refrain from code refactoring and focus only on adding types to your modules.

The fifth step is to enable "noImplicitAny," which will enforce that all types are known and defined, providing a better TypeScript experience for your project.

During the migration, you can use the `@ts-check` directive, which enables TypeScript type checking in a JavaScript file. This directive provides a loose version of type checking and can be initially used to identify issues in JavaScript files. When `@ts-check` is included in a file, TypeScript will try to deduce definitions using JSDoc-style comments. However, consider using JSDoc annotations only at a very early stage of the migration.

Consider keeping the default value of `noEmitOnError` in your tsconfig.json as false. This will allow you to output JavaScript source code even if errors are reported.

Exploring the Type System

The TypeScript Language Service

The TypeScript Language Service, also known as tsserver, offers various features such as error reporting, diagnostics, compile-on-save, renaming, go to definition, completion lists, signature help, and more. It is primarily used by integrated development environments (IDEs) to provide IntelliSense support. It seamlessly integrates with Visual Studio Code and is utilized by tools like Conquer of Completion (Coc).

Developers can leverage a dedicated API and create their own custom language service plugins to enhance the TypeScript editing experience. This can be particularly useful for implementing special linting features or enabling auto-completion for a custom templating language.

An example of a real-world custom plugin is "typescript-styled-plugin", which provides syntax error reporting and IntelliSense support for CSS properties in styled components.

For more information and quick start guides, you can refer to the official TypeScript Wiki on GitHub: https://github.com/microsoft/TypeScript/wiki/

Structural Typing

TypeScript is based on a structural type system. This means that the compatibility and equivalence of types are determined by the type's actual structure or definition, rather than its name or place of declaration, as in nominative type systems like C# or C.

TypeScript's structural type system was designed based on how JavaScript's dynamic duck typing system works during runtime.

The following example is valid TypeScript code. As you can observe, "X" and "Y" have the same member "a," even though they have different declaration names. The types are determined by their structures, and in this case, since the structures are the same, they are compatible and valid.

```
type X = {
    a: string;
};
type Y = {
    a: string;
};
const x: X = { a: 'a' };
const y: Y = x; // Valid
```

TypeScript Fundamental Comparison Rules

The TypeScript comparison process is recursive and executed on types nested at any level.

A type "X" is compatible with "Y" if "Y" has at least the same members as "X".

```
type X = {
    a: string;
};
const y = { a: 'A', b: 'B' }; // Valid, as it has at least the same
members as X
const r: X = y;
```

Function parameters are compared by types, not by their names:

```
type X = (a: number) => void;
type Y = (a: number) => void;
let x: X = (j: number) => undefined;
let y: Y = (k: number) => undefined;
y = x; // Valid
x = y; // Valid
```

Function return types must be the same:

```
type X = (a: number) => undefined;
type Y = (a: number) => number;
let x: X = (a: number) => undefined;
let y: Y = (a: number) => 1;
y = x; // Invalid
x = y; // Invalid
```

The return type of a source function must be a subtype of the return type of a target function:

```
let x = () => ({ a: 'A' });
let y = () => ({ a: 'A', b: 'B' });
x = y; // Valid
y = x; // Invalid member b is missing
```

Discarding function parameters is allowed, as it is a common practice in JavaScript, for instance using "Array.prototype.map()":

```
[1, 2, 3].map((element, _index, _array) => element + 'x');
```

Therefore, the following type declarations are completely valid:

```
type X = (a: number) => undefined;
type Y = (a: number, b: number) => undefined;
```

```
let x: X = (a: number) => undefined;
let y: Y = (a: number) => undefined; // Missing b parameter
y = x; // Valid
```

Any additional optional parameters of the source type are valid:

```
type X = (a: number, b?: number, c?: number) => undefined;
type Y = (a: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // Valid
x = y; //Valid
```

Any optional parameters of the target type without corresponding parameters in the source type are valid and not an error:

```
type X = (a: number) => undefined;
type Y = (a: number, b?: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // Valid
x = y; // Valid
```

The rest parameter is treated as an infinite series of optional parameters:

```
type X = (a: number, ...rest: number[]) => undefined;
let x: X = a => undefined; //valid
```

Functions with overloads are valid if the overload signature is compatible with its implementation signature:

```
function x(a: string): void;
function x(a: string, b: number): void;
function x(a: string, b?: number): void {
    console.log(a, b);
}
x('a'); // Valid
x('a', 1); // Valid

function y(a: string): void; // Invalid, not compatible with
implementation signature
function y(a: string, b: number): void;
function y(a: string, b: number): void {
    console.log(a, b);
}
y('a');
y('a', 1);
```

Function parameter comparison succeeds if the source and target parameters are

assignable to supertypes or subtypes (bivariance).

```
// Supertype
class X {
    a: string;
    constructor(value: string) {
        this.a = value;
    }
}
// Subtype
class Y extends X {}
// Subtype
class Z extends X {}

type GetA = (x: X) => string;
const getA: GetA = x => x.a;

// Bivariance does accept supertypes
console.log(getA(new X('x'))); // Valid
console.log(getA(new Y('Y'))); // Valid
console.log(getA(new Z('z'))); // Valid
```

Enums are comparable and valid with numbers and vice versa, but comparing Enum values from different Enum types is invalid.

```
enum X {
    A,
    B,
}
enum Y {
    A,
    B,
    C,
}
const xa: number = X.A; // Valid
const ya: Y = 0; // Valid
X.A === Y.A; // Invalid
```

Instances of a class are subject to a compatibility check for their private and protected members:

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}

class Y {
```

```
    private a: string;
    constructor(value: string) {
        this.a = value;
    }
}

let x: X = new Y('y'); // Invalid
```

The comparison check does not take into consideration the different inheritance hierarchy, for instance:

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
class Y extends X {
    public a: string;
    constructor(value: string) {
        super(value);
        this.a = value;
    }
}
class Z {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
let x: X = new X('x');
let y: Y = new Y('y');
let z: Z = new Z('z');
x === y; // Valid
x === z; // Valid even if z is from a different inheritance hierarchy
```

Generics are compared using their structures based on the resulting type after applying the generic parameter, only the final result is compared as a non-generic type.

```
interface X<T> {
    a: T;
}
let x: X<number> = { a: 1 };
let y: X<string> = { a: 'a' };
x === y; // Invalid as the type argument is used in the final structure
```

```
interface X<T> {}
const x: X<number> = 1;
const y: X<string> = 'a';
x === y; // Valid as the type argument is not used in the final structure
```

When generics do not have their type argument specified, all the unspecified arguments are treated as types with "any":

```
type X = <T>(x: T) => T;
type Y = <K>(y: K) => K;
let x: X = x => x;
let y: Y = y => y;
x = y; // Valid
```

Remember:

```
let a: number = 1;
let b: number = 2;
a = b; // Valid, everything is assignable to itself

let c: any;
c = 1; // Valid, all types are assignable to any

let d: unknown;
d = 1; // Valid, all types are assignable to unknown

let e: unknown;
let e1: unknown = e; // Valid, unknown is only assignable to itself and
any
let e2: any = e; // Valid
let e3: number = e; // Invalid

let f: never;
f = 1; // Invalid, nothing is assignable to never

let g: void;
let g1: any;
g = 1; // Invalid, void is not assignable to or from anything expect any
g = g1; // Valid
```

Please note that when "strictNullChecks" is enabled, "null" and "undefined" are treated similarly to "void"; otherwise, they are similar to "never".

Types as Sets

In TypeScript, a type is a set of possible values. This set is also referred to as the domain of the type. Each value of a type can be viewed as an element in a set. A type establishes the constraints that every element in the set must satisfy to be considered a member of that set. The primary task of TypeScript is to check and verify whether one set is a subset of another.

TypeScript supports various types of sets:

Set term	TypeScript	Notes
Empty set	never	"never" contains anything apart itself
Single element set	undefined / null / literal type	
Finite set	boolean / union	
Infinite set	string / number / object	
Universal set	any / unknown	Every element is a member of "any" and every set is a subset of it / "unknown" is a type-safe counterpart of "any"

Here few examples:

TypeScript	Set term	Example
never	∅ (empty set)	const x: never = 'x'; // Error: Type 'string' is not assignable to type 'never'
Literal type	Single element set	type X = 'X'; type Y = 7;
Value assignable to T	Value ∈ T (member of)	type XY = 'X' | 'Y'; const x: XY = 'X';
T1 assignable to T2	T1 ⊆ T2 (subset of)	type XY = 'X' | 'Y'; const x: XY = 'X'; const j: XY = 'J'; // Type '"J"' is not assignable to type 'XY'.
T1 extends T2	T1 ⊆ T2 (subset of)	type X = 'X' extends string ? true : false;
T1 | T2	T1 ∪ T2 (union)	type XY = 'X' | 'Y'; type JK = 1 | 2;
T1 & T2	T1 ∩ T2 (intersection)	type X = { a: string } type Y = { b: string } type XY = X & Y const x: XY = { a: 'a', b: 'b' }
unknown	Universal set	const x: unknown = 1

An union, (T1 | T2) creates a wider set (both):

```
type X = {
    a: string;
};
type Y = {
    b: string;
};
type XY = X | Y;
const r: XY = { a: 'a', b: 'x' }; // Valid
```

An intersection, (T1 & T2) create a narrower set (only shared):

```
type X = {
    a: string;
```

```
};
type Y = {
    a: string;
    b: string;
};
type XY = X & Y;
const r: XY = { a: 'a' }; // Invalid
const j: XY = { a: 'a', b: 'b' }; // Valid
```

The `extends` keyword could be considered as a "subset of" in this context. It sets a constraint for a type. The extends used with a generic, take the generic as an infinite set and it will constrain it to a more specific type. Please note that `extends` has nothing to do with hierarchy in a OOP sense (there is no this concept in TypeScript). TypeScript works with sets and does not have a strict hierarchy, infact, as in the example below, two types could overlap without either being a subtype of the other type (TypeScript considers the structure, shape of the objects).

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
interface Z extends Y {
    c: string;
}
const z: Z = { a: 'a', b: 'b', c: 'c' };
interface X1 {
    a: string;
}
interface Y1 {
    a: string;
    b: string;
}
interface Z1 {
    a: string;
    b: string;
    c: string;
}
const z1: Z1 = { a: 'a', b: 'b', c: 'c' };

const r: Z1 = z; // Valid
```

Assign a type: Type Declarations and Type

Assertions

A type can be assigned in different ways in TypeScript:

Type Declaration

In the following example, we use x: X (": Type") to declare a type for the variable x.

```
type X = {
    a: string;
};

// Type declaration
const x: X = {
    a: 'a',
};
```

If the variable is not in the specified format, TypeScript will report an error. For instance:

```
type X = {
    a: string;
};

const x: X = {
    a: 'a',
    b: 'b', // Error: Object literal may only specify known properties
};
```

Type Assertion

It is possible to add an assertion by using the `as` keyword. This tells the compiler that the developer has more information about a type and silences any errors that may occur.

For example:

```
type X = {
    a: string;
};
const x = {
    a: 'a',
    b: 'b',
} as X;
```

In the above example, the object x is asserted to have the type X using the as keyword. This informs the TypeScript compiler that the object conforms to the specified type, even though it has an additional property b not present in the type definition.

Type assertions are useful in situations where a more specific type needs to be specified, especially when working with the DOM. For instance:

```
const myInput = document.getElementById('my_input') as HTMLInputElement;
```

Here, the type assertion as HTMLInputElement is used to tell TypeScript that the result of getElementById should be treated as an HTMLInputElement. Type assertions can also be used to remap keys, as shown in the example below with template literals:

```
type J<Type> = {
    [Property in keyof Type as `prefix_${string &
        Property}`]: () => Type[Property];
};
type X = {
    a: string;
    b: number;
};
type Y = J<X>;
```

In this example, the type `J<Type>` uses a mapped type with a template literal to remap the keys of Type. It creates new properties with a "prefix_" added to each key, and their corresponding values are functions returning the original property values.

It is worth noting that when using a type assertion, TypeScript will not execute excess property checking. Therefore, it is generally preferable to use a Type Declaration when the structure of the object is known in advance.

Ambient Declarations

Ambient declarations are files that describe types for JavaScript code, they have a file name format as `.d.ts.`. They are usually imported and used to annotate existing JavaScript libraries or to add types to existing JS files in your project.

Many common libraries types can be found at:
https://github.com/DefinitelyTyped/DefinitelyTyped/

and can be installed using:

```
npm install --save-dev @types/library-name
```

For your defined Ambient Declarations, you can import using the "triple-slash" reference:

```
/// <reference path="./library-types.d.ts" />
```

You can use Ambient Declarations even within JavaScript files using `// @ts-check`.

The `declare` keyword enables type definitions for existing JavaScript code without importing it, serving as a placeholder for types from another file or globally.

Property Checking and Excess Property Checking

TypeScript is based on a structural type system but excess property checking is a property of TypeScript which allows it to check whether an object has the exact properties specified in the type.

Excess Property Checking is performed when assigning object literals to variables or when passing them as arguments to the function's excess property, for instance.

```
type X = {
    a: string;
};
const y = { a: 'a', b: 'b' };
const x: X = y; // Valid because structural typing
const w: X = { a: 'a', b: 'b' }; // Invalid because excess property
checking
```

Weak Types

A type is considered weak when it contains nothing but a set of all-optional properties:

```
type X = {
    a?: string;
    b?: string;
};
```

TypeScript considers an error to assign anything to a weak type when there is no overlap, for instance, the following throws an error:

```
type Options = {
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;

fn({ c: 'c' }); // Invalid
```

Although not recommended, if needed, it is possible to bypass this check by using type assertion:

```
type Options = {
    a?: string;
    b?: string;
};
```

```
const fn = (options: Options) => undefined;
fn({ c: 'c' } as Options); // Valid
```

Or by adding `unknown` to the index signature to the weak type:

```
type Options = {
    [prop: string]: unknown;
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;
fn({ c: 'c' }); // Valid
```

Strict Object Literal Checking (Freshness)

Strict object literal checking, sometimes referred to as "freshness", is a feature in TypeScript that helps catch excess or misspelled properties that would otherwise go unnoticed in normal structural type checks.

When creating an object literal, the TypeScript compiler considers it "fresh." If the object literal is assigned to a variable or passed as a parameter, TypeScript will throw an error if the object literal specifies properties that do not exist in the target type.

However, "freshness" disappears when an object literal is widened or a type assertion is used.

Here are some examples to illustrate:

```
type X = { a: string };
type Y = { a: string; b: string };

let x: X;
x = { a: 'a', b: 'b' }; // Freshness check: Invalid assignment
var y: Y;
y = { a: 'a', bx: 'bx' }; // Freshness check: Invalid assignment

const fn = (x: X) => console.log(x.a);

fn(x);
fn(y); // Widening: No errors, structurally type compatible

fn({ a: 'a', bx: 'b' }); // Freshness check: Invalid argument

let c: X = { a: 'a' };
let d: Y = { a: 'a', b: '' };
c = d; // Widening: No Freshness check
```

Type Inference

TypeScript can infer types when no annotation is provided during:

- Variable initialization.
- Member initialization.
- Setting defaults for parameters.
- Function return type.

For example:

```
let x = 'x'; // The type inferred is string
```

The TypeScript compiler analyzes the value or expression and determines its type based on the available information.

More Advanced Inferences

When multiple expressions are used in type inference, TypeScript looks for the "best common types." For instance:

```
let x = [1, 'x', 1, null]; // The type inferred is: (string | number |
null)[]
```

If the compiler cannot find the best common types, it returns a union type. For example:

```
let x = [new RegExp('x'), new Date()]; // Type inferred is: (RegExp |
Date)[]
```

TypeScript utilizes "contextual typing" based on the variable's location to infer types. In the following example, the compiler knows that `e` is of type `MouseEvent` because of the `click` event type defined in the lib.d.ts file, which contains ambient declarations for various common JavaScript constructs and the DOM:

```
window.addEventListener('click', function (e) {}); // The inferred type of
e is MouseEvent
```

Type Widening

Type widening is the process in which TypeScript assigns a type to a variable initialized when no type annotation was provided. It allows narrow to wider types but not vice versa. In the following example:

```
let x = 'x'; // TypeScript infers as string, a wide type
let y: 'y' | 'x' = 'y'; // y types is a union of literal types
y = x; // Invalid Type 'string' is not assignable to type '"x" | "y"'.
```

TypeScript assigns `string` to `x` based on the single value provided during initialization (`x`), this is an example of widening.

TypeScript provides ways to have control of the widening process, for instance using "const".

Const

Using the `const` keyword when declaring a variable results in a narrower type inference in TypeScript.

For example:

```
const x = 'x'; // TypeScript infers the type of x as 'x', a narrower type
let y: 'y' | 'x' = 'y';
y = x; // Valid: The type of x is inferred as 'x'
```

By using `const` to declare the variable x, its type is narrowed to the specific literal value 'x'. Since the type of x is narrowed, it can be assigned to the variable y without any error. The reason the type can be inferred is because `const` variables cannot be reassigned, so their type can be narrowed down to a specific literal type, in this case, the literal type 'x'.

Const Modifier on Type Parameters

From version 5.0 of TypeScript, it is possible to specify the `const` attribute on a generic type parameter. This allows for inferring the most precise type possible. Let's see an example without using `const`:

```
function identity<T>(value: T) {
    // No const here
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // Type infered is: { a:
string; b: string; }
```

As you can see, the properties `a` and `b` are inferred with a type of `string` .

Now, let's see the difference with the `const` version:

```
function identity<const T>(value: T) {
    // Using const modifier on type parameters
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // Type infered is: { a: "a";
```

```
b: "b"; }
```

Now we can see that the properties `a` and `b` are inferred as `const`, so `a` and `b` are treated as string literals rather than just `string` types.

Const assertion

This feature allows you to declare a variable with a more precise literal type based on its initialization value, signifying to the compiler that the value should be treated as an immutable literal. Here are a few examples:

On a single property:

```
const v = {
    x: 3 as const,
};
v.x = 3;
```

On an entire object:

```
const v = {
    x: 1,
    y: 2,
} as const;
```

This can be particularly useful when defining the type for a tuple:

```
const x = [1, 2, 3]; // number[]
const y = [1, 2, 3] as const; // Tuple of readonly [1, 2, 3]
```

Explicit Type Annotation

We can be specific and pass a type, in the following example property `x` is of type `number`:

```
const v = {
    x: 1, // Inferred type: number (widening)
};
v.x = 3; // Valid
```

We can make the type annotation more specific by using a union of literal types:

```
const v: { x: 1 | 2 | 3 } = {
    x: 1, // x is now a union of literal types: 1 | 2 | 3
};
v.x = 3; // Valid
v.x = 100; // Invalid
```

Type Narrowing

Type Narrowing is the process in TypeScript where a general type is narrowed down to a more specific type. This occurs when TypeScript analyzes the code and determines that certain conditions or operations can refine the type information.

Narrowing types can occur in different ways, including:

Conditions

By using conditional statements, such as `if` or `switch`, TypeScript can narrow down the type based on the outcome of the condition. For example:

```
let x: number | undefined = 10;

if (x !== undefined) {
    x += 100; // The type is number, which had been narrowed by the
condition
}
```

Throwing or returning

Throwing an error or returning early from a branch can be used to help TypeScript narrow down a type. For example:

```
let x: number | undefined = 10;

if (x === undefined) {
    throw 'error';
}
x += 100;
```

Other ways to narrow down types in TypeScript include:

- `instanceof` operator: Used to check if an object is an instance of a specific class.
- `in` operator: Used to check if a property exists in an object.
- `typeof` operator: Used to check the type of a value at runtime.
- Built-in functions like `Array.isArray()`: Used to check if a value is an array.

Discriminated Union

Using a "Discriminated Union" is a pattern in TypeScript where an explicit "tag" is added to objects to distinguish between different types within a union. This pattern is also referred to as a "tagged union." In the following example, the "tag" is represented by the property "type":

```
type A = { type: 'type_a'; value: number };
```

```
type B = { type: 'type_b'; value: string };

const x = (input: A | B): string | number => {
    switch (input.type) {
        case 'type_a':
            return input.value + 100; // type is A
        case 'type_b':
            return input.value + 'extra'; // type is B
    }
};
```

User-Defined Type Guards

In cases where TypeScript is unable to determine a type, it is possible to write a helper function known as a "user-defined type guard." In the following example, we will utilize a Type Predicate to narrow down the type after applying certain filtering:

```
const data = ['a', null, 'c', 'd', null, 'f'];

const r1 = data.filter(x => x != null); // The type is (string | null)[],
TypeScript was not able to infer the type properly

const isValid = (item: string | null): item is string => item !== null; //
Custom type guard

const r2 = data.filter(isValid); // The type is fine now string[], by
using the predicate type guard we were able to narrow the type
```

Primitive Types

TypeScript supports 7 primitive types. A primitive data type refers to a type that is not an object and does not have any methods associated with it. In TypeScript, all primitive types are immutable, meaning their values cannot be changed once they are assigned.

string

The `string` primitive type stores textual data, and the value is always double or single-quoted.

```
const x: string = 'x';
const y: string = 'y';
```

Strings can span multiple lines if surrounded by the backtick (`) character:

```
let sentence: string = `xxx,
   yyy`;
```

boolean

The `boolean` data type in TypeScript stores a binary value, either `true` or `false`.

```
const isReady: boolean = true;
```

number

A `number` data type in TypeScript is represented with a 64-bit floating point value. A `number` type can represent integers and fractions. TypeScript also supports hexadecimal, binary, and octal, for instance:

```
const decimal: number = 10;
const hexadecimal: number = 0xa00d; // Hexadecimal starts with 0x
const binary: number = 0b1010; // Binary starts with 0b
const octal: number = 0o633; // Octal starts with 0o
```

bigInt

A `bigInt` represents numeric values that are very large (253 – 1) and cannot be represented with a `number`.

A `bigInt` can be created by calling the built-in function `BigInt()` or by adding `n` to the

end of any integer numeric literal:

```
const x: bigint = BigInt(9007199254740991);
const y: bigint = 9007199254740991n;
```

Notes:

- `bigInt` values cannot be mixed with `number` and cannot be used with built-in `Math`, they must be coerced to the same type.
- `bigInt` values are available only if target configuration is ES2020 or higher.

Symbol

Symbols are unique identifiers that can be used as property keys in objects to prevent naming conflicts.

```
type Obj = {
    [sym: symbol]: number;
};

const a = Symbol('a');
const b = Symbol('b');
let obj: Obj = {};
obj[a] = 123;
obj[b] = 456;

console.log(obj[a]); // 123
console.log(obj[b]); // 456
```

null and undefined

`null` and `undefined` types both represent no value or the absence of any value.

The `undefined` type means the value is not assigned or initialized or indicates an unintentional absence of value.

The `null` type means that we know that the field does not have a value, so value is unavailable, it indicates an intentional absence of value.

Array

An `array` is a data type that can store multiple values of the same type or not. It can be defined using the following syntax:

```
const x: string[] = ['a', 'b'];
const y: Array<string> = ['a', 'b'];
const j: Array<string | number> = ['a', 1, 'b', 2]; // Union
```

TypeScript supports readonly arrays using the following syntax:

```
const x: readonly string[] = ['a', 'b']; // Readonly modifier
const y: ReadonlyArray<string> = ['a', 'b'];
const j: ReadonlyArray<string | number> = ['a', 1, 'b', 2];
j.push('x'); // Invalid
```

TypeScript supports tuple and readonly tuple:

```
const x: [string, number] = ['a', 1];
const y: readonly [string, number] = ['a', 1];
```

any

The `any` data type represents literally "any" value, it is the default value when TypeScript cannot infer the type or is not specified.

When using `any` TypeScript compiler skips the type checking so there is no type safety when `any` is being used. Generally do not use `any` to silence the compiler when an error occurs, instead focus on fixing the error as with using `any` it is possible to break contracts and we lose the benefits of TypeScript autocomplete.

The `any` type could be useful during a gradual migration from JavaScript to TypeScript, as it can silence the compiler.

For new projects use TypeScript configuration `noImplicitAny` which enables TypeScript to issue errors where `any` is used or inferred.

The `any`type is usually a source of errors which can mask real problems with your types. Avoid using it as much as possible.

Type Annotations

On variables declared using `var`, `let` and `const`, it is possible to optionally add a type:

```
const x: number = 1;
```

TypeScript does a good job of inferring types, especially when simple one, so these declarations in most cases are not necessary.

On functions is possible to add type annotations to parameters:

```
function sum(a: number, b: number) {
    return a + b;
}
```

The following is an example using a anonymous functions (so called lambda function):

```
const sum = (a: number, b: number) => a + b;
```

These annotation can be avoided when a default value for a parameter is present:

```
const sum = (a = 10, b: number) => a + b;
```

Return type annotations can be added to functions:

```
const sum = (a = 10, b: number): number => a + b;
```

This is useful especially for more complex functions as writing expliciting the return type before an implementation can help better think about the function.

Generally consider annotating type signatures but not the body local variables and add types always to object literals.

Optional Properties

An object can specify Optional Properties by adding a question mark `?` to the end of the property name:

```
type X = {
    a: number;
    b?: number; // Optional
};
```

It is possible to specify a default value when a property is optional"

```
type X = {
    a: number;
    b?: number;
};
const x = ({ a, b = 100 }: X) => a + b;
```

Readonly Properties

Is it possible to prevent writing on a property by using the modifier `readonly`which makes sure that the property cannot be re-written but does not provide any guarantee of total immutability:

```
interface Y {
    readonly a: number;
}

type X = {
    readonly a: number;
};

type J = Readonly<{
    a: number;
}>;

type K = {
    readonly [index: number]: string;
};
```

Index Signatures

In TypeScript we can use as index signature `string`, `number`, and `symbol`:

```
type K = {
    [name: string | number]: string;
};
const k: K = { x: 'x', 1: 'b' };
console.log(k['x']);
console.log(k[1]);
console.log(k['1']); // Same result as k[1]
```

Please note that JavaScript automatically converts an index with `number` to an index with `string` so `k[1]` or `k["1"]` return the same value.

Extending Types

It is possible to extend an `interface` (copy members from another type):

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
```

It is also possible to extend from multiple types:

```
interface A {
    a: string;
}
interface B {
    b: string;
}
interface Y extends A, B {
    y: string;
}
```

The `extends` keyword works only on interfaces and classes, for types use an intersection:

```
type A = {
    a: number;
};
type B = {
    b: number;
};
type C = A & B;
```

It is possible to extend a type using an inference but not vice versa:

```
type A = {
    a: string;
};
interface B extends A {
    b: string;
}
```

Literal Types

A Literal Type is a single element set from a collective type, it defines a very exact value that is a JavaScript primitive.

Literal Types in TypeScript are numbers, strings, and booleans.

Example of literals:

```
const a = 'a'; // String literal type
const b = 1; // Numeric literal type
const c = true; // Boolean literal type
```

String, Numeric, and Boolean Literal Types are used in the union, type guard, and type aliases. In the following example you can see a type alias union, `0` can be the only value specified and not any other string:

```
type 0 = 'a' | 'b' | 'c';
```

Literal Inference

Literal Inference is a feature in TypeScript that allows the type of a variable or parameter to be inferred based on its value.

In the following example we can see that TypeScript considers `x` a literal type as the value cannot be changed any time later, when instead `y` is inferred as string as it can be modified any time later.

```
const x = 'x'; // Literal type of 'x', because this value cannot be
changed
let y = 'y'; // Type string, as we can change this value
```

In the following example we can see that `o.x` was inferred as a `string` (and not a literal of `a`) as TypeScript considers that the value can be changed any time later.

```
type X = 'a' | 'b';

let o = {
    x: 'a', // This is a wider string
};

const fn = (x: X) => `${x}-foo`;

console.log(fn(o.x)); // Argument of type 'string' is not assignable to
parameter of type 'X'
```

As you can see the code throws an error when passing `o.x` to `fn` as X is a narrower type.

We can solve this issue by using type assertion using `const` or the `X` type:

```
let o = {
    x: 'a' as const,
};
```

or:

```
let o = {
    x: 'a' as X,
};
```

strictNullChecks

`strictNullChecks` is a TypeScript compiler option that enforces strict null checking. When this option is enabled, variables and parameters can only be assigned `null` or `undefined` if they have been explicitly declared to be of that type using the union type `null | undefined`. If a variable or parameter is not explicitly declared as nullable, TypeScript will generate an error to prevent potential runtime errors.

Enums

In TypeScript, an `enum` is a set of named constant values.

```
enum Color {
    Red = '#ff0000',
    Green = '#00ff00',
    Blue = '#0000ff',
}
```

Enums can be defined in different ways:

Numeric enums

In TypeScript, a Numeric Enum is an Enum where each constant is assigned a numeric value, starting from 0 by default.

```
enum Size {
    Small, // value starts from 0
    Medium,
    Large,
}
```

It is possible to specify custom values by explicitly assigning them:

```
enum Size {
    Small = 10,
    Medium,
    Large,
}
console.log(Size.Medium); // 11
```

String enums

In TypeScript, a String enum is an Enum where each constant is assigned a string value.

```
enum Language {
    English = 'EN',
    Spanish = 'ES',
}
```

Note: TypeScript allows the usage of heterogeneous Enums where string and numeric members can coexist.

Constant enums

A constant enum in TypeScript is a special type of Enum where all the values are known at compile time and are inlined wherever the enum is used, resulting in more efficient code.

```
const enum Language {
    English = 'EN',
    Spanish = 'ES',
}
console.log(Language.English);
```

Will be compiled into:

```
console.log('EN' /* Language.English */);
```

Notes: Const Enums have hardcoded values, erasing the Enum, which can be more efficient in self-contained libraries but is generally not desirable. Also, Const enums cannot have computed members.

Reverse mapping

In TypeScript, reverse mappings in Enums refer to the ability to retrieve the Enum member name from its value. By default, Enum members have forward mappings from name to value, but reverse mappings can be created by explicitly setting values for each member. Reverse mappings are useful when you need to look up an Enum member by its value, or when you need to iterate over all the Enum members. Note that only numeric enums members will generate reverse mappings, while String Enum members do not get a reverse mapping generated at all.

The following enum:

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    F = 'fail',
}
```

Compiles to:

```
'use strict';
var Grade;
(function (Grade) {
    Grade[(Grade['A'] = 90)] = 'A';
    Grade[(Grade['B'] = 80)] = 'B';
    Grade[(Grade['C'] = 70)] = 'C';
    Grade['F'] = 'fail';
```

```
})(Grade || (Grade = {}));
```

Therefore, mapping values to keys works for numeric enum members, but not for string enum members:

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    F = 'fail',
}
const myGrade = Grade.A;
console.log(Grade[myGrade]); // A
console.log(Grade[90]); // A

const failGrade = Grade.F;
console.log(failGrade); // fail
console.log(Grade[failGrade]); // Element implicitly has an 'any' type
because index expression is not of type 'number'.
```

Ambient enums

An ambient enum in TypeScript is a type of Enum that is defined in a declaration file (*.d.ts) without an associated implementation. It allows you to define a set of named constants that can be used in a type-safe way across different files without having to import the implementation details in each file.

Computed and constant members

In TypeScript, a computed member is a member of an Enum that has a value calculated at runtime, while a constant member is a member whose value is set at compile-time and cannot be changed during runtime. Computed members are allowed in regular Enums, while constant members are allowed in both regular and const enums.

```
// Constant members
enum Color {
    Red = 1,
    Green = 5,
    Blue = Red + Green,
}
console.log(Color.Blue); // 6 generation at compilation time
```

```
// Computed members
enum Color {
    Red = 1,
    Green = Math.pow(2, 2),
    Blue = Math.floor(Math.random() * 3) + 1,
```

```
}
console.log(Color.Blue); // random number generated at run time
```

Enums are denoted by unions comprising their member types. The values of each member can be determined through constant or non-constant expressions, with members possessing constant values being assigned literal types. To illustrate, consider the declaration of type E and its subtypes E.A, E.B, and E.C. In this case, E represents the union E.A | E.B | E.C.

```
const identity = (value: number) => value;

enum E {
    A = 2 * 5, // Numeric literal
    B = 'bar', // String literal
    C = identity(42), // Opaque computed
}

console.log(E.C); //42
```

Narrowing

TypeScript narrowing is the process of refining the type of a variable within a conditional block. This is useful when working with union types, where a variable can have more than one type.

TypeScript recognizes several ways to narrow the type:

typeof type guards

The typeof type guard is one specific type guard in TypeScript that checks the type of a variable based on its built-in JavaScript type.

```
const fn = (x: number | string) => {
    if (typeof x === 'number') {
        return x + 1; // x is number
    }
    return -1;
};
```

Truthiness narrowing

Truthiness narrowing in TypeScript works by checking whether a variable is truthy or falsy to narrow its type accordingly.

```
const toUpperCase = (name: string | null) => {
    if (name) {
        return name.toUpperCase();
    } else {
        return null;
    }
};
```

Equality narrowing

Equality narrowing in TypeScript works by checking whether a variable is equal to a specific value or not, to narrow its type accordingly.

It is used in conjunction with `switch` statements and equality operators such as `===`, `!==`, `==`, and `!=` to narrow down types.

```
const checkStatus = (status: 'success' | 'error') => {
    switch (status) {
        case 'success':
```

```
            return true;
        case 'error':
            return null;
    }
};
```

In Operator narrowing

The `in` Operator narrowing in TypeScript is a way to narrow the type of a variable based on whether a property exists within the variable's type.

```
type Dog = {
    name: string;
    breed: string;
};

type Cat = {
    name: string;
    likesCream: boolean;
};

const getAnimalType = (pet: Dog | Cat) => {
    if ('breed' in pet) {
        return 'dog';
    } else {
        return 'cat';
    }
};
```

instanceof narrowing

The `instanceof` operator narrowing in TypeScript is a way to narrow the type of a variable based on its constructor function, by checking if an object is an instance of a certain class or interface.

```
class Square {
    constructor(public width: number) {}
}
class Rectangle {
    constructor(
        public width: number,
        public height: number
    ) {}
}
function area(shape: Square | Rectangle) {
    if (shape instanceof Square) {
        return shape.width * shape.width;
```

```
    } else {
        return shape.width * shape.height;
    }
}
const square = new Square(5);
const rectangle = new Rectangle(5, 10);
console.log(area(square)); // 25
console.log(area(rectangle)); // 50
```

Assignments

TypeScript narrowing using assignments is a way to narrow the type of a variable based on the value assigned to it. When a variable is assigned a value, TypeScript infers its type based on the assigned value, and it narrows the type of the variable to match the inferred type.

```
let value: string | number;
value = 'hello';
if (typeof value === 'string') {
    console.log(value.toUpperCase());
}
value = 42;
if (typeof value === 'number') {
    console.log(value.toFixed(2));
}
```

Control Flow Analysis

Control Flow Analysis in TypeScript is a way to statically analyze the code flow to infer the types of variables, allowing the compiler to narrow the types of those variables as needed, based on the results of the analysis.

Prior to TypeScript 4.4, code flow analysis would only be applied to code within an if statement, but from TypeScript 4.4, it can also be applied to conditional expressions and discriminant property accesses indirectly referenced through const variables.

For example:

```
const f1 = (x: unknown) => {
    const isString = typeof x === 'string';
    if (isString) {
        x.length;
    }
};

const f2 = (
    obj: { kind: 'foo'; foo: string } | { kind: 'bar'; bar: number }
) => {
    const isFoo = obj.kind === 'foo';
    if (isFoo) {
        obj.foo;
    } else {
        obj.bar;
    }
};
```

Some examples where narrowing does not occur:

```
const f1 = (x: unknown) => {
    let isString = typeof x === 'string';
    if (isString) {
        x.length; // Error, no narrowing because isString it is not const
    }
};

const f6 = (
    obj: { kind: 'foo'; foo: string } | { kind: 'bar'; bar: number }
) => {
    const isFoo = obj.kind === 'foo';
    obj = obj;
    if (isFoo) {
        obj.foo; // Error, no narrowing because obj is assigned in
function body
    }
```

```
};
```

Notes: Up to five levels of indirection are analyzed in conditional expressions.

Type Predicates

Type Predicates in TypeScript are functions that return a boolean value and are used to narrow the type of a variable to a more specific type.

```
const isString = (value: unknown): value is string => typeof value ===
'string';

const foo = (bar: unknown) => {
    if (isString(bar)) {
        console.log(bar.toUpperCase());
    } else {
        console.log('not a string');
    }
};
```

Discriminated Unions

Discriminated Unions in TypeScript are a type of union type that uses a common property, known as the discriminant, to narrow down the set of possible types for the union.

```
type Square = {
    kind: 'square'; // Discriminant
    size: number;
};

type Circle = {
    kind: 'circle'; // Discriminant
    radius: number;
};

type Shape = Square | Circle;

const area = (shape: Shape) => {
    switch (shape.kind) {
        case 'square':
            return Math.pow(shape.size, 2);
        case 'circle':
            return Math.PI * Math.pow(shape.radius, 2);
    }
};

const square: Square = { kind: 'square', size: 5 };
const circle: Circle = { kind: 'circle', radius: 2 };

console.log(area(square)); // 25
console.log(area(circle)); // 12.566370614359172
```

The never Type

When a variable is narrowed to a type that cannot contain any values, the TypeScript compiler will infer that the variable must be of the `never` type. This is because The never Type represents a value that can never be produced.

```
const printValue = (val: string | number) => {
    if (typeof val === 'string') {
        console.log(val.toUpperCase());
    } else if (typeof val === 'number') {
        console.log(val.toFixed(2));
    } else {
        // val has type never here because it can never be anything other
than a string or a number
        const neverVal: never = val;
        console.log(`Unexpected value: ${neverVal}`);
    }
};
```

Exhaustiveness checking

Exhaustiveness checking is a feature in TypeScript that ensures all possible cases of a discriminated union are handled in a `switch` statement or an `if` statement.

```
type Direction = 'up' | 'down';

const move = (direction: Direction) => {
    switch (direction) {
        case 'up':
            console.log('Moving up');
            break;
        case 'down':
            console.log('Moving down');
            break;
        default:
            const exhaustiveCheck: never = direction;
            console.log(exhaustiveCheck); // This line will never be
executed
    }
};
```

The `never` type is used to ensure that the default case is exhaustive and that TypeScript will raise an error if a new value is added to the Direction type without being handled in the switch statement.

Object Types

In TypeScript, object types describe the shape of an object. They specify the names and types of the object's properties, as well as whether those properties are required or optional.

In TypeScript, you can define object types in two primary ways:

Interface which defines the shape of an object by specifying the names, types, and optionality of its properties.

```
interface User {
    name: string;
    age: number;
    email?: string;
}
```

Type alias, similar to an interface, defines the shape of an object. However, it can also create a new custom type that is based on an existing type or a combination of existing types. This includes defining union types, intersection types, and other complex types.

```
type Point = {
    x: number;
    y: number;
};
```

It also possible to define a type anonymously:

```
const sum = (x: { a: number; b: number }) => x.a + x.b;
console.log(sum({ a: 5, b: 1 }));
```

Tuple Type (Anonymous)

A Tuple Type is a type that represents an array with a fixed number of elements and their corresponding types. A tuple type enforces a specific number of elements and their respective types in a fixed order. Tuple types are useful when you want to represent a collection of values with specific types, where the position of each element in the array has a specific meaning.

```
type Point = [number, number];
```

Named Tuple Type (Labeled)

Tuple types can include optional labels or names for each element. These labels are for readability and tooling assistance, and do not affect the operations you can perform with them.

```
type T = string;
type Tuple1 = [T, T];
type Tuple2 = [a: T, b: T];
type Tuple3 = [a: T, T]; // Named Tuple plus Anonymous Tuple
```

Fixed Length Tuple

A Fixed Length Tuple is a specific type of tuple that enforces a fixed number of elements of specific types, and disallows any modifications to the length of the tuple once it is defined.

Fixed Length Tuples are useful when you need to represent a collection of values with a specific number of elements and specific types, and you want to ensure that the length and types of the tuple cannot be changed inadvertently.

```
const x = [10, 'hello'] as const;
x.push(2); // Error
```

Union Type

A Union Type is a type that represents a value that can be one of several types. Union Types are denoted using the `|` symbol between each possible type.

```
let x: string | number;
x = 'hello'; // Valid
x = 123; // Valid
```

Intersection Types

An Intersection Type is a type that represents a value that has all the properties of two or more types. Intersection Types are denoted using the `&` symbol between each type.

```
type X = {
    a: string;
};

type Y = {
    b: string;
};

type J = X & Y; // Intersection

const j: J = {
    a: 'a',
    b: 'b',
};
```

Type Indexing

Type indexing refers to the ability to define types that can be indexed by a key that is not known in advance, using an index signature to specify the type for properties that are not explicitly declared.

```
type Dictionary<T> = {
    [key: string]: T;
};
const myDict: Dictionary<string> = { a: 'a', b: 'b' };
console.log(myDict['a']); // Returns a
```

Type from Value

Type from Value in TypeScript refers to the automatic inference of a type from a value or expression through type inference.

```
const x = 'x'; // TypeScript can automatically infer that the type of the message variable is string
```

Type from Func Return

Type from Func Return refers to the ability to automatically infer the return type of a function based on its implementation. This allows TypeScript to determine the type of the value returned by the function without explicit type annotations.

```
const add = (x: number, y: number) => x + y; // TypeScript can infer that
the return type of the function is a number
```

Type from Module

Type from Module refers to the ability to use a module's exported values to automatically infer their types. When a module exports a value with a specific type, TypeScript can use that information to automatically infer the type of that value when it is imported into another module.

```
// calc.ts
export const add = (x: number, y: number) => x + y;
// index.ts
import { add } from 'calc';
const r = add(1, 2); // r is number
```

Mapped Types

Mapped Types in TypeScript allow you to create new types based on an existing type by transforming each property using a mapping function. By mapping existing types, you can create new types that represent the same information in a different format. To create a mapped type, you access the properties of an existing type using the `keyof` operator and then alter them to produce a new type. In the following example:

```
type MyMappedType<T> = {
    [P in keyof T]: T[P][];
};
type MyType = {
    foo: string;
    bar: number;
};
type MyNewType = MyMappedType<MyType>;
const x: MyNewType = {
    foo: ['hello', 'world'],
    bar: [1, 2, 3],
};
```

we define MyMappedType to map over T's properties, creating a new type with each property as an array of its original type. Using this, we create MyNewType to represent the same info as MyType, but with each property as an array.

Mapped Type Modifiers

Mapped Type Modifiers in TypeScript enable the transformation of properties within an existing type:

- `readonly` or `+readonly`: This renders a property in the mapped type as read-only.
- `-readonly`: This allows a property in the mapped type to be mutable.
- `?`: This designates a property in the mapped type as optional.

Examples:

```
type ReadOnly<T> = { readonly [P in keyof T]: T[P] }; // All properties marked as read-only

type Mutable<T> = { -readonly [P in keyof T]: T[P] }; // All properties marked as mutable

type MyPartial<T> = { [P in keyof T]?: T[P] }; // All properties marked as optional
```

Conditional Types

Conditional Types are a way to create a type that depends on a condition, where the type to be created is determined based on the result of the condition. They are defined using the `extends` keyword and a ternary operator to conditionally choose between two types.

```
type IsArray<T> = T extends any[] ? true : false;

const myArray = [1, 2, 3];
const myNumber = 42;

type IsMyArrayAnArray = IsArray<typeof myArray>; // Type true
type IsMyNumberAnArray = IsArray<typeof myNumber>; // Type false
```

Distributive Conditional Types

Distributive Conditional Types are a feature that allow a type to be distributed over a union of types, by applying a transformation to each member of the union individually. This can be especially useful when working with mapped types or higher-order types.

```
type Nullable<T> = T extends any ? T | null : never;
type NumberOrBool = number | boolean;
type NullableNumberOrBool = Nullable<NumberOrBool>; // number | boolean |
null
```

infer Type Inference in Conditional Types

The `infer` keyword is used in conditional types to infer (extract) the type of a generic parameter from a type that depends on it. This allows you to write more flexible and reusable type definitions.

```
type ElementType<T> = T extends (infer U)[] ? U : never;
type Numbers = ElementType<number[]>; // number
type Strings = ElementType<string[]>; // string
```

Predefined Conditional Types

In TypeScript, Predefined Conditional Types are built-in conditional types provided by the language. They are designed to perform common type transformations based on the characteristics of a given type.

`Exclude<UnionType, ExcludedType>`: This type removes all the types from Type that are assignable to ExcludedType.

`Extract<Type, Union>`: This type extracts all the types from Union that are assignable to Type.

`NonNullable<Type>`: This type removes null and undefined from Type.

`ReturnType<Type>`: This type extracts the return type of a function Type.

`Parameters<Type>`: This type extracts the parameter types of a function Type.

`Required<Type>`: This type makes all properties in Type required.

`Partial<Type>`: This type makes all properties in Type optional.

`Readonly<Type>`: This type makes all properties in Type readonly.

Template Union Types

Template union types can be used to merge and manipulate text inside the type system for instance:

```
type Status = 'active' | 'inactive';
type Products = 'p1' | 'p2';
type ProductId = `id-${Products}-${Status}`; // "id-p1-active" | "id-p1-
inactive" | "id-p2-active" | "id-p2-inactive"
```

Any type

The `any` type is a special type (universal supertype) that can be used to represent any type of value (primitives, objects, arrays, functions, errors, symbols). It is often used in situations where the type of a value is not known at compile time, or when working with values from external APIs or libraries that do not have TypeScript typings.

By utilizing `any` type, you are indicating to the TypeScript compiler that values should be represented without any limitations. In order to maximizing type safety in your code consider the following:

- Limit the usage of `any` to specific cases where the type is truly unknown.
- Do not return `any` types from a function as you will lose type safety in the code using that function weakening your type safety.
- Instead of `any` use `@ts-ignore` if you need to silence the compiler.

```
let value: any;
value = true; // Valid
value = 7; // Valid
```

Unknown type

In TypeScript, the `unknown` type represents a value that is of an unknown type. Unlike `any` type, which allows for any type of value, `unknown` requires a type check or assertion before it can be used in a specific way so no operations are permitted on an `unknown` without first asserting or narrowing to a more specific type.

The `unknown` type is only assignable to any type and the `unknown` type itself, it is a type-safe alternative to `any`.

```
let value: unknown;

let value1: unknown = value; // Valid
let value2: any = value; // Valid
let value3: boolean = value; // Invalid
let value4: number = value; // Invalid
```

```
const add = (a: unknown, b: unknown): number | undefined =>
    typeof a === 'number' && typeof b === 'number' ? a + b : undefined;
console.log(add(1, 2)); // 3
console.log(add('x', 2)); // undefined
```

Void type

The `void` type is used to indicate that a function does not return a value.

```
const sayHello = (): void => {
    console.log('Hello!');
};
```

Never type

The `never` type represents values that never occur. It is used to denote functions or expressions that never return or throw an error.

For instance an infinite loop:

```
const infiniteLoop = (): never => {
    while (true) {
        // do something
    }
};
```

Throwing an error:

```
const throwError = (message: string): never => {
    throw new Error(message);
};
```

The `never` type is useful in ensuring type safety and catching potential errors in your code. It helps TypeScript analyze and infer more precise types when used in combination with other types and control flow statements, for instance:

```
type Direction = 'up' | 'down';
const move = (direction: Direction): void => {
    switch (direction) {
        case 'up':
            // move up
            break;
        case 'down':
            // move down
            break;
        default:
            const exhaustiveCheck: never = direction;
            throw new Error(`Unhandled direction: ${exhaustiveCheck}`);
    }
};
```

Interface and Type

Common Syntax

In TypeScript, interfaces define the structure of objects, specifying the names and types of properties or methods that an object must have. The common syntax for defining an interface in TypeScript is as follows:

```
interface InterfaceName {
    property1: Type1;
    // ...
    method1(arg1: ArgType1, arg2: ArgType2): ReturnType;
    // ...
}
```

Similarly for type definition:

```
type TypeName = {
    property1: Type1;
    // ...
    method1(arg1: ArgType1, arg2: ArgType2): ReturnType;
    // ...
};
```

`interface InterfaceName` or `type TypeName`: Defines the name of the interface. `property1`: `Type1`: Specifies the properties of the interface along with their corresponding types. Multiple properties can be defined, each separated by a semicolon. `method1(arg1: ArgType1, arg2: ArgType2): ReturnType;`: Specifies the methods of the interface. Methods are defined with their names, followed by a parameter list in parentheses and the return type. Multiple methods can be defined, each separated by a semicolon.

Example interface:

```
interface Person {
    name: string;
    age: number;
    greet(): void;
}
```

Example of type:

```
type TypeName = {
    property1: string;
    method1(arg1: string, arg2: string): string;
};
```

In TypeScript, types are used to define the shape of data and enforce type checking. There are several common syntaxes for defining types in TypeScript, depending on the specific use case. Here are some examples:

Basic Types

```
let myNumber: number = 123; // number type
let myBoolean: boolean = true; // boolean type
let myArray: string[] = ['a', 'b']; // array of strings
let myTuple: [string, number] = ['a', 123]; // tuple
```

Objects and Interfaces

```
const x: { name: string; age: number } = { name: 'Simon', age: 7 };
```

Union and Intersection Types

```
type MyType = string | number; // Union type
let myUnion: MyType = 'hello'; // Can be a string
myUnion = 123; // Or a number

type TypeA = { name: string };
type TypeB = { age: number };
type CombinedType = TypeA & TypeB; // Intersection type
let myCombined: CombinedType = { name: 'John', age: 25 }; // Object with
both name and age properties
```

Built-in Type Primitives

TypeScript has several built-in type primitives that can be used to define variables, function parameters, and return types:

- `number`: Represents numeric values, including integers and floating-point numbers.
- `string`: Represents textual data
- `boolean`: Represents logical values, which can be either true or false.
- `null`: Represents the absence of a value.
- `undefined`: Represents a value that has not been assigned or has not been defined.
- `symbol`: Represents a unique identifier. Symbols are typically used as keys for object properties.
- `bigint`: Represents arbitrary-precision integers.
- `any`: Represents a dynamic or unknown type. Variables of type any can hold values of any type, and they bypass type checking.
- `void`: Represents the absence of any type. It is commonly used as the return type of functions that do not return a value.
- `never`: Represents a type for values that never occur. It is typically used as the return type of functions that throw an error or enter an infinite loop.

Common Built-in JS Objects

TypeScript is a superset of JavaScript, it includes all the commonly used built-in JavaScript objects. You can find an extensive list of these objects on the Mozilla Developer Network (MDN) documentation website: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Here is a list of some commonly used built-in JavaScript objects:

- Function
- Object
- Boolean
- Error
- Number
- BigInt
- Math
- Date
- String
- RegExp
- Array
- Map
- Set
- Promise
- Intl

Overloads

Function overloads in TypeScript allow you to define multiple function signatures for a single function name, enabling you to define functions that can be called in multiple ways. Here's an example:

```
// Overloads
function sayHi(name: string): string;
function sayHi(names: string[]): string[];

// Implementation
function sayHi(name: unknown): unknown {
    if (typeof name === 'string') {
        return `Hi, ${name}!`;
    } else if (Array.isArray(name)) {
        return name.map(name => `Hi, ${name}!`);
    }
    throw new Error('Invalid value');
}

sayHi('xx'); // Valid
sayHi(['aa', 'bb']); // Valid
```

Here's another example of using function overloads within a `class`:

```
class Greeter {
    message: string;

    constructor(message: string) {
        this.message = message;
    }

    // overload
    sayHi(name: string): string;
    sayHi(names: string[]): ReadonlyArray<string>;

    // implementation
    sayHi(name: unknown): unknown {
        if (typeof name === 'string') {
            return `${this.message}, ${name}!`;
        } else if (Array.isArray(name)) {
            return name.map(name => `${this.message}, ${name}!`);
        }
        throw new Error('value is invalid');
    }
}
console.log(new Greeter('Hello').sayHi('Simon'));
```

Merging and Extension

Merging and extension refer to two different concepts related to working with types and interfaces.

Merging allows you to combine multiple declarations of the same name into a single definition, for example, when you define an interface with the same name multiple times:

```
interface X {
    a: string;
}

interface X {
    b: number;
}

const person: X = {
    a: 'a',
    b: 7,
};
```

Extension refers to the ability to extend or inherit from existing types or interfaces to create new ones. It is a mechanism to add additional properties or methods to an existing type without modifying its original definition. Example:

```
interface Animal {
    name: string;
    eat(): void;
}

interface Bird extends Animal {
    sing(): void;
}

const dog: Bird = {
    name: 'Bird 1',
    eat() {
        console.log('Eating');
    },
    sing() {
        console.log('Singing');
    },
};
```

Differences between Type and Interface

Declaration merging (augmentation):

Interfaces support declaration merging, which means that you can define multiple interfaces with the same name, and TypeScript will merge them into a single interface with the combined properties and methods. On the other hand, types do not support declaration merging. This can be helpful when you want to add extra functionality or customize existing types without modifying the original definitions or patching missing or incorrect types.

```
interface A {
    x: string;
}
interface A {
    y: string;
}
const j: A = {
    x: 'xx',
    y: 'yy',
};
```

Extending other types/interfaces:

Both types and interfaces can extend other types/interfaces, but the syntax is different. With interfaces, you use the `extends` keyword to inherit properties and methods from other interfaces. However, an interface cannot extend a complex type like a union type.

```
interface A {
    x: string;
    y: number;
}
interface B extends A {
    z: string;
}
const car: B = {
    x: 'x',
    y: 123,
    z: 'z',
};
```

For types, you use the & operator to combine multiple types into a single type (intersection).

```
interface A {
    x: string;
    y: number;
```

```
}

type B = A & {
    j: string;
};

const c: B = {
    x: 'x',
    y: 123,
    j: 'j',
};
```

Union and Intersection Types:

Types are more flexible when it comes to defining Union and Intersection Types. With the `type` keyword, you can easily create union types using the `|` operator and intersection types using the `&` operator. While interfaces can also represent union types indirectly, they don't have built-in support for intersection types.

```
type Department = 'dep-x' | 'dep-y'; // Union

type Person = {
    name: string;
    age: number;
};

type Employee = {
    id: number;
    department: Department;
};

type EmployeeInfo = Person & Employee; // Intersection
```

Example with interfaces:

```
interface A {
    x: 'x';
}
interface B {
    y: 'y';
}

type C = A | B; // Union of interfaces
```

Class

Class Common Syntax

The `class` keyword is used in TypeScript to define a class. Below, you can see an example:

```
class Person {
    private name: string;
    private age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    public sayHi(): void {
        console.log(
            `Hello, my name is ${this.name} and I am ${this.age} years
old.`
        );
    }
}
```

The `class` keyword is used to define a class named "Person".

The class has two private properties: name of type `string` and age of type `number`.

The constructor is defined using the `constructor` keyword. It takes name and age as parameters and assigns them to the corresponding properties.

The class has a `public` method named sayHi that logs a greeting message.

To create an instance of a class in TypeScript, you can use the `new` keyword followed by the class name, followed by parentheses `()`. For instance:

```
const myObject = new Person('John Doe', 25);
myObject.sayHi(); // Output: Hello, my name is John Doe and I am 25 years
old.
```

Constructor

Constructors are special methods within a class that are used to initialize the object's properties when an instance of the class is created.

```
class Person {
    public name: string;
    public age: number;
```

```
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    sayHello() {
        console.log(
            `Hello, my name is ${this.name} and I'm ${this.age} years
old.`
        );
    }
}

const john = new Person('Simon', 17);
john.sayHello();
```

It is possible to overload a constructor using the following syntax:

```
type Sex = 'm' | 'f';

class Person {
    name: string;
    age: number;
    sex: Sex;

    constructor(name: string, age: number, sex?: Sex);
    constructor(name: string, age: number, sex: Sex) {
        this.name = name;
        this.age = age;
        this.sex = sex ?? 'm';
    }
}

const p1 = new Person('Simon', 17);
const p2 = new Person('Alice', 22, 'f');
```

In TypeScript, it is possible to define multiple constructor overloads, but you can have only one implementation that must be compatible with all the overloads, this can be achieved by using an optional parameter.

```
class Person {
    name: string;
    age: number;

    constructor();
    constructor(name: string);
    constructor(name: string, age: number);
    constructor(name?: string, age?: number) {
        this.name = name ?? 'Unknown';
        this.age = age ?? 0;
```

```
    }

    displayInfo() {
        console.log(`Name: ${this.name}, Age: ${this.age}`);
    }
}

const person1 = new Person();
person1.displayInfo(); // Name: Unknown, Age: 0

const person2 = new Person('John');
person2.displayInfo(); // Name: John, Age: 0

const person3 = new Person('Jane', 25);
person3.displayInfo(); // Name: Jane, Age: 25
```

Private and Protected Constructors

In TypeScript, constructors can be marked as private or protected, which restricts their accessibility and usage.

Private Constructors: Can be called only within the class itself. Private constructors are often used in scenarios where you want to enforce a singleton pattern or restrict the creation of instances to a factory method within the class

Protected Constructors: Protected constructors are useful when you want to create a base class that should not be instantiated directly but can be extended by subclasses.

```
class BaseClass {
    protected constructor() {}
}

class DerivedClass extends BaseClass {
    private value: number;

    constructor(value: number) {
        super();
        this.value = value;
    }
}

// Attempting to instantiate the base class directly will result in an
error
// const baseObj = new BaseClass(); // Error: Constructor of class
'BaseClass' is protected.

// Create an instance of the derived class
const derivedObj = new DerivedClass(10);
```

Access Modifiers

Access Modifiers `private`, `protected`, and `public` are used to control the visibility and accessibility of class members, such as properties and methods, in TypeScript classes. These modifiers are essential for enforcing encapsulation and establishing boundaries for accessing and modifying the internal state of a class.

The `private` modifier restricts access to the class member only within the containing class.

The `protected` modifier allows access to the class member within the containing class and its derived classes.

The `public` modifier provides unrestricted access to the class member, allowing it to be accessed from anywhere."

Get and Set

Getters and setters are special methods that allow you to define custom access and modification behavior for class properties. They enable you to encapsulate the internal state of an object and provide additional logic when getting or setting the values of properties. In TypeScript, getters and setters are defined using the `get` and `set` keywords respectively. Here's an example:

```
class MyClass {
    private _myProperty: string;

    constructor(value: string) {
        this._myProperty = value;
    }
    get myProperty(): string {
        return this._myProperty;
    }
    set myProperty(value: string) {
        this._myProperty = value;
    }
}
```

Auto-Accessors in Classes

TypeScript version 4.9 adds support for auto-accessors, a forthcoming ECMAScript feature. They resemble class properties but are declared with the "accessor" keyword.

```
class Animal {
    accessor name: string;
```

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

Auto-accessors are "de-sugared" into private `get` and `set` accessors, operating on an inaccessible property.

```
class Animal {
    #__name: string;

    get name() {
        return this.#__name;
    }
    set name(value: string) {
        this.#__name = name;
    }

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

this

In TypeScript, the `this` keyword refers to the current instance of a class within its methods or constructors. It allows you to access and modify the properties and methods of the class from within its own scope. It provides a way to access and manipulate the internal state of an object within its own methods.

```
class Person {
    private name: string;
    constructor(name: string) {
        this.name = name;
    }
    public introduce(): void {
        console.log(`Hello, my name is ${this.name}.`);
    }
}

const person1 = new Person('Alice');
person1.introduce(); // Hello, my name is Alice.
```

Parameter Properties

Parameter properties allow you to declare and initialize class properties directly within the

constructor parameters avoiding boilerplate code, example:

```
class Person {
    constructor(
        private name: string,
        public age: number
    ) {
        // The "private" and "public" keywords in the constructor
        // automatically declare and initialize the corresponding class
properties.
    }
    public introduce(): void {
        console.log(
            `Hello, my name is ${this.name} and I am ${this.age} years
old.`
        );
    }
}
const person = new Person('Alice', 25);
person.introduce();
```

Abstract Classes

Abstract Classes are used in TypeScript mainly for inheritance, they provide a way to define common properties and methods that can be inherited by subclasses. This is useful when you want to define common behavior and enforce that subclasses implement certain methods. They provide a way to create a hierarchy of classes where the abstract base class provides a shared interface and common functionality for the subclasses.

```
abstract class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    abstract makeSound(): void;
}

class Cat extends Animal {
    makeSound(): void {
        console.log(`${this.name} meows.`);
    }
}

const cat = new Cat('Whiskers');
cat.makeSound(); // Output: Whiskers meows.
```

With Generics

Classes with generics allow you to define reusable classes which can work with different types.

```
class Container<T> {
    private item: T;

    constructor(item: T) {
        this.item = item;
    }

    getItem(): T {
        return this.item;
    }

    setItem(item: T): void {
        this.item = item;
    }
}

const container1 = new Container<number>(42);
console.log(container1.getItem()); //  42

const container2 = new Container<string>('Hello');
container2.setItem('World');
console.log(container2.getItem()); // World
```

Decorators

Decorators provide a mechanism to add metadata, modify behavior, validate, or extend the functionality of the target element. They are functions that execute at runtime. Multiple decorators can be applied to a declaration.

Decorators are experimental features, and the following examples are only compatible with TypeScript version 5 or above using ES6.

For TypeScript versions prior to 5, they should be enabled using the `experimentalDecorators` property in your `tsconfig.json` or by using `--experimentalDecorators` in your command line (but the following example won't work).

Some of the common use cases for decorators include:

- Watching property changes.
- Watching method calls.
- Adding extra properties or methods.

- Runtime validation.
- Automatic serialization and deserialization.
- Logging.
- Authorization and authentication.
- Error guarding.

Note: Decorators for version 5 do not allow decorating parameters.

Types of decorators:

Class Decorators

Class Decorators are useful for extending an existing class, such as adding properties or methods, or collecting instances of a class. In the following example, we add a `toString` method that converts the class into a string representation.

```
type Constructor<T = {}> = new (...args: any[]) => T;

function toString<Class extends Constructor>(
    Value: Class,
    context: ClassDecoratorContext<Class>
) {
    return class extends Value {
        constructor(...args: any[]) {
            super(...args);
            console.log(JSON.stringify(this));
            console.log(JSON.stringify(context));
        }
    };
}

@toString
class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    greet() {
        return 'Hello, ' + this.name;
    }
}
const person = new Person('Simon');
/* Logs:
{"name":"Simon"}
{"kind":"class","name":"Person"}
```

```
*/
```

Property Decorator

Property decorators are useful for modifying the behavior of a property, such as changing the initialization values. In the following code, we have a script that sets a property to always be in uppercase:

```
function upperCase<T>(
    target: undefined,
    context: ClassFieldDecoratorContext<T, string>
) {
    return function (this: T, value: string) {
        return value.toUpperCase();
    };
}

class MyClass {
    @upperCase
    prop1 = 'hello!';
}

console.log(new MyClass().prop1); // Logs: HELLO!
```

Method Decorator

Method decorators allow you to change or enhance the behavior of methods. Below is an example of a simple logger:

```
function log<This, Args extends any[], Return>(
    target: (this: This, ...args: Args) => Return,
    context: ClassMethodDecoratorContext<
        This,
        (this: This, ...args: Args) => Return
    >
) {
    const methodName = String(context.name);

    function replacementMethod(this: This, ...args: Args): Return {
        console.log(`LOG: Entering method '${methodName}'.`);
        const result = target.call(this, ...args);
        console.log(`LOG: Exiting method '${methodName}'.`);
        return result;
    }

    return replacementMethod;
}

class MyClass {
    @log
```

```
    sayHello() {
        console.log('Hello!');
    }
}

new MyClass().sayHello();
```

It logs:

```
LOG: Entering method 'sayHello'.
Hello!
LOG: Exiting method 'sayHello'.
```

Getter and Setter Decorators

Getter and setter decorators allow you to change or enhance the behavior of class accessors. They are useful, for instance, for validating property assignments. Here's a simple example for a getter decorator:

```
function range<This, Return extends number>(min: number, max: number) {
    return function (
        target: (this: This) => Return,
        context: ClassGetterDecoratorContext<This, Return>
    ) {
        return function (this: This): Return {
            const value = target.call(this);
            if (value < min || value > max) {
                throw 'Invalid';
            }
            Object.defineProperty(this, context.name, {
                value,
                enumerable: true,
            });
            return value;
        };
    };
}

class MyClass {
    private _value = 0;

    constructor(value: number) {
        this._value = value;
    }
    @range(1, 100)
    get getValue(): number {
        return this._value;
    }
}
```

```
const obj = new MyClass(10);
console.log(obj.getValue); // Valid: 10

const obj2 = new MyClass(999);
console.log(obj2.getValue); // Throw: Invalid!
```

Decorator Metadata

Decorator Metadata simplifies the process for decorators to apply and utilize metadata in any class. They can access a new metadata property on the context object, which can serve as a key for both primitives and objects. Metadata information can be accessed on the class via `Symbol.metadata`.

Metadata can be used for various purposes, such as debugging, serialization, or dependency injection with decorators.

```
//@ts-ignore
Symbol.metadata ??= Symbol('Symbol.metadata'); // Simple polify

type Context =
    | ClassFieldDecoratorContext
    | ClassAccessorDecoratorContext
    | ClassMethodDecoratorContext; // Context contains property metadata:
DecoratorMetadata

function setMetadata(_target: any, context: Context) {
    // Set the metadata object with a primitive value
    context.metadata[context.name] = true;
}

class MyClass {
    @setMetadata
    a = 123;

    @setMetadata
    accessor b = 'b';

    @setMetadata
    fn() {}
}

const metadata = MyClass[Symbol.metadata]; // Get metadata information

console.log(JSON.stringify(metadata)); //
{"bar":true,"baz":true,"foo":true}
```

Inheritance

Inheritance refers to the mechanism by which a class can inherit properties and methods from another class, known as the base class or superclass. The derived class, also called the child class or subclass, can extend and specialize the functionality of the base class by adding new properties and methods or overriding existing ones.

```
class Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    speak(): void {
        console.log('The animal makes a sound');
    }
}

class Dog extends Animal {
    breed: string;

    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }

    speak(): void {
        console.log('Woof! Woof!');
    }
}

// Create an instance of the base class
const animal = new Animal('Generic Animal');
animal.speak(); // The animal makes a sound

// Create an instance of the derived class
const dog = new Dog('Max', 'Labrador');
dog.speak(); // Woof! Woof!"
```

TypeScript does not support multiple inheritance in the traditional sense and instead allows inheritance from a single base class. TypeScript supports multiple interfaces. An interface can define a contract for the structure of an object, and a class can implement multiple interfaces. This allows a class to inherit behavior and structure from multiple sources.

```
interface Flyable {
    fly(): void;
}
```

```
interface Swimmable {
    swim(): void;
}

class FlyingFish implements Flyable, Swimmable {
    fly() {
        console.log('Flying...');
    }

    swim() {
        console.log('Swimming...');
    }
}

const flyingFish = new FlyingFish();
flyingFish.fly();
flyingFish.swim();
```

The `class` keyword in TypeScript, similar to JavaScript, is often referred to as syntactic sugar. It was introduced in ECMAScript 2015 (ES6) to offer a more familiar syntax for creating and working with objects in a class-based manner. However, it's important to note that TypeScript, being a superset of JavaScript, ultimately compiles down to JavaScript, which remains prototype-based at its core.

Statics

TypeScript has static members. To access the static members of a class, you can use the class name followed by a dot, without the need to create an object.

```
class OfficeWorker {
    static memberCount: number = 0;

    constructor(private name: string) {
        OfficeWorker.memberCount++;
    }
}

const w1 = new OfficeWorker('James');
const w2 = new OfficeWorker('Simon');
const total = OfficeWorker.memberCount;
console.log(total); // 2
```

Property initialization

There are several ways how you can initialize properties for a class in TypeScript:

Inline:

In the following example these initial values will be used when an instance of the class is created.

```
class MyClass {
    property1: string = 'default value';
    property2: number = 42;
}
```

In the constructor:

```
class MyClass {
    property1: string;
    property2: number;

    constructor() {
        this.property1 = 'default value';
        this.property2 = 42;
    }
}
```

Using constructor parameters:

```
class MyClass {
    constructor(
        private property1: string = 'default value',
        public property2: number = 42
    ) {
        // There is no need to assign the values to the properties
explicitly.
    }
    log() {
        console.log(this.property2);
    }
}
const x = new MyClass();
x.log();
```

Method overloading

Method overloading allows a class to have multiple methods with the same name but different parameter types or a different number of parameters. This allows us to call a method in different ways based on the arguments passed.

```
class MyClass {
    add(a: number, b: number): number; // Overload signature 1
    add(a: string, b: string): string; // Overload signature 2

    add(a: number | string, b: number | string): number | string {
```

```
        if (typeof a === 'number' && typeof b === 'number') {
            return a + b;
        }
        if (typeof a === 'string' && typeof b === 'string') {
            return a.concat(b);
        }
        throw new Error('Invalid arguments');
    }
}

const r = new MyClass();
console.log(r.add(10, 5)); // Logs 15
```

Generics

Generics allow you to create reusable components and functions that can work with multiple types. With generics, you can parameterize types, functions, and interfaces, allowing them to operate on different types without explicitly specifying them beforehand.

Generics allow you to make code more flexible and reusable.

Generic Type

To define a generic type, you use angle brackets (`<>`) to specify the type parameters, for instance:

```
function identity<T>(arg: T): T {
    return arg;
}
const a = identity('x');
const b = identity(123);

const getLen = <T,>(data: ReadonlyArray<T>) => data.length;
const len = getLen([1, 2, 3]);
```

Generic Classes

Generics can be applied also to classes, in this way they can work with multiple types by using type parameters. This is useful to create reusable class definitions that can operate on different data types while maintaining type safety.

```
class Container<T> {
    private item: T;

    constructor(item: T) {
        this.item = item;
    }

    getItem(): T {
        return this.item;
    }
}

const numberContainer = new Container<number>(123);
console.log(numberContainer.getItem()); // 123

const stringContainer = new Container<string>('hello');
console.log(stringContainer.getItem()); // hello
```

Generic Constraints

Generic parameters can be constrained using the `extends` keyword followed by a type or interface that the type parameter must satisfy.

In the following example T it is must containing a properly `length` in order to be valid:

```
const printLen = <T extends { length: number }>(value: T): void => {
    console.log(value.length);
};

printLen('Hello'); // 5
printLen([1, 2, 3]); // 3
printLen({ length: 10 }); // 10
printLen(123); // Invalid
```

An interesting feature of generic introduced in version 3.4 RC is Higher order function type inference which introduced propagated generic type arguments:

```
declare function pipe<A extends any[], B, C>(
    ab: (...args: A) => B,
    bc: (b: B) => C
): (...args: A) => C;

declare function list<T>(a: T): T[];
declare function box<V>(x: V): { value: V };

const listBox = pipe(list, box); // <T>(a: T) => { value: T[] }
const boxList = pipe(box, list); // <V>(x: V) => { value: V }[]
```

This functionality allows more easily typed safe pointfree style programming which is common in functional programming.

Generic contextual narrowing

Contextual narrowing for generics is the mechanism in TypeScript that allows the compiler to narrow down the type of a generic parameter based on the context in which it is used, it is useful when working with generic types in conditional statements:

```
function process<T>(value: T): void {
    if (typeof value === 'string') {
        // Value is narrowed down to type 'string'
        console.log(value.length);
    } else if (typeof value === 'number') {
        // Value is narrowed down to type 'number'
        console.log(value.toFixed(2));
```

```
    }
}

process('hello'); // 5
process(3.14159); // 3.14
```

Erased Structural Types

In TypeScript, objects do not have to match a specific, exact type. For instance, if we create an object that fulfills an interface's requirements, we can utilize that object in places where that interface is required, even if there was no explicit connection between them. Example:

```
type NameProp1 = {
    prop1: string;
};

function log(x: NameProp1) {
    console.log(x.prop1);
}

const obj = {
    prop2: 123,
    prop1: 'Origin',
};

log(obj); // Valid
```

Namespacing

In TypeScript, namespaces are used to organize code into logical containers, preventing naming collisions and providing a way to group related code together. The usage of the `export` keywords allows access to the namespace in "outside" modules.

```
export namespace MyNamespace {
    export interface MyInterface1 {
        prop1: boolean;
    }
    export interface MyInterface2 {
        prop2: string;
    }
}

const a: MyNamespace.MyInterface1 = {
    prop1: true,
};
```

Symbols

Symbols are a primitive data type that represents an immutable value which is guaranteed to be globally unique throughout the lifetime of the program.

Symbols can be used as keys for object properties and provide a way to create non-enumerable properties.

```
const key1: symbol = Symbol('key1');
const key2: symbol = Symbol('key2');

const obj = {
    [key1]: 'value 1',
    [key2]: 'value 2',
};

console.log(obj[key1]); // value 1
console.log(obj[key2]); // value 2
```

In WeakMaps and WeakSets, symbols are now permissible as keys.

Triple-Slash Directives

Triple-slash directives are special comments that provide instructions to the compiler about how to process a file. These directives begin with three consecutive slashes (`///`) and are typically placed at the top of a TypeScript file and have no effects on the runtime behavior.

Triple-slash directives are used to reference external dependencies, specify module loading behavior, enable/disable certain compiler features, and more. Few examples:

Referencing a declaration file:

```
/// <reference path="path/to/declaration/file.d.ts" />
```

Indicate the module format:

```
/// <amd|commonjs|system|umd|es6|es2015|none>
```

Enable compiler options, in the following example strict mode:

```
/// <strict|noImplicitAny|noUnusedLocals|noUnusedParameters>
```

Type Manipulation

Creating Types from Types

Is it possible to create new types composing, manipulating or transforming existing types.

Intersection Types (&):

Allow you to combine multiple types into a single type:

```
type A = { foo: number };
type B = { bar: string };
type C = A & B; // Intersection of A and B
const obj: C = { foo: 42, bar: 'hello' };
```

Union Types (|):

Allow you to define a type that can be one of several types:

```
type Result = string | number;
const value1: Result = 'hello';
const value2: Result = 42;
```

Mapped Types:

Allow you to transform the properties of an existing type to create new type:

```
type Mutable<T> = {
    readonly [P in keyof T]: T[P];
};
type Person = {
    name: string;
    age: number;
};
type ImmutablePerson = Mutable<Person>; // Properties become read-only
```

Conditional types:

Allow you to create types based on some conditions:

```
type ExtractParam<T> = T extends (param: infer P) => any ? P : never;
type MyFunction = (name: string) => number;
type ParamType = ExtractParam<MyFunction>; // string
```

Indexed Access Types

In TypeScript is it possible to access and manipulate the types of properties within another type using an index, `Type[Key]`.

```
type Person = {
    name: string;
    age: number;
};

type AgeType = Person['age']; // number
```

```
type MyTuple = [string, number, boolean];
type MyType = MyTuple[2]; // boolean
```

Utility Types

Several built-in utility types can be used to manipulate types, below a list of the most common used:

Awaited<T>

Constructs a type recursively unwrap Promises.

```
type A = Awaited<Promise<string>>; // string
```

Partial<T>

Constructs a type with all properties of T set to optional.

```
type Person = {
    name: string;
    age: number;
};

type A = Partial<Person>; // { name?: string | undefined; age?: number |
undefined; }
```

Required<T>

Constructs a type with all properties of T set to required.

```
type Person = {
    name?: string;
    age?: number;
};
```

```
type A = Required<Person>; // { name: string; age: number; }
```

Readonly<T>

Constructs a type with all properties of T set to readonly.

```
type Person = {
    name: string;
    age: number;
};

type A = Readonly<Person>;

const a: A = { name: 'Simon', age: 17 };
a.name = 'John'; // Invalid
```

Record<K, T>

Constructs a type with a set of properties K of type T.

```
type Product = {
    name: string;
    price: number;
};

const products: Record<string, Product> = {
    apple: { name: 'Apple', price: 0.5 },
    banana: { name: 'Banana', price: 0.25 },
};

console.log(products.apple); // { name: 'Apple', price: 0.5 }
```

Pick<T, K>

Constructs a type by picking the specified properties K from T.

```
type Product = {
    name: string;
    price: number;
};

type Price = Pick<Product, 'price'>; // { price: number; }
```

Omit<T, K>

Constructs a type by omitting the specified properties K from T.

```
type Product = {
    name: string;
    price: number;
```

```
};

type Name = Omit<Product, 'price'>; // { name: string; }
```

Exclude<T, U>

Constructs a type by excluding all values of type U from T.

```
type Union = 'a' | 'b' | 'c';
type MyType = Exclude<Union, 'a' | 'c'>; // b
```

Extract<T, U>

Constructs a type by extracting all values of type U from T.

```
type Union = 'a' | 'b' | 'c';
type MyType = Extract<Union, 'a' | 'c'>; // a | c
```

NonNullable<T>

Constructs a type by excluding null and undefined from T.

```
type Union = 'a' | null | undefined | 'b';
type MyType = NonNullable<Union>; // 'a' | 'b'
```

Parameters<T>

Extracts the parameter types of a function type T.

```
type Func = (a: string, b: number) => void;
type MyType = Parameters<Func>; // [a: string, b: number]
```

ConstructorParameters<T>

Extracts the parameter types of a constructor function type T.

```
class Person {
    constructor(
        public name: string,
        public age: number
    ) {}
}
type PersonConstructorParams = ConstructorParameters<typeof Person>; //
[name: string, age: number]
const params: PersonConstructorParams = ['John', 30];
const person = new Person(...params);
console.log(person); // Person { name: 'John', age: 30 }
```

ReturnType<T>

Extracts the return type of a function type T.

```
type Func = (name: string) => number;
type MyType = ReturnType<Func>; // number
```

InstanceType<T>

Extracts the instance type of a class type T.

```
class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    sayHello() {
        console.log(`Hello, my name is ${this.name}!`);
    }
}

type PersonInstance = InstanceType<typeof Person>;

const person: PersonInstance = new Person('John');

person.sayHello(); // Hello, my name is John!
```

ThisParameterType<T>

Extracts the type of 'this' parameter from a function type T.

```
interface Person {
    name: string;
    greet(this: Person): void;
}
type PersonThisType = ThisParameterType<Person['greet']>; // Person
```

OmitThisParameter<T>

Removes the 'this' parameter from a function type T.

```
function capitalize(this: String) {
    return this[0].toUpperCase + this.substring(1).toLowerCase();
}

type CapitalizeType = OmitThisParameter<typeof capitalize>; // () =>
string
```

ThisType<T>

Servers as a market for a contextual `this` type.

```
type Logger = {
    log: (error: string) => void;
};

let helperFunctions: { [name: string]: Function } & ThisType<Logger> = {
    hello: function () {
        this.log('some error'); // Valid as "log" is a part of "this".
        this.update(); // Invalid
    },
};
```

Uppercase<T>

Make uppercase the name of the input type T.

```
type MyType = Uppercase<'abc'>; // "ABC"
```

Lowercase<T>

Make lowercase the name of the input type T.

```
type MyType = Lowercase<'ABC'>; // "abc"
```

Capitalize<T>

Capitalize the name of the input type T.

```
type MyType = Capitalize<'abc'>; // "Abc"
```

Uncapitalize<T>

Uncapitalize the name of the input type T.

```
type MyType = Uncapitalize<'Abc'>; // "abc"
```

NoInfer<T>

NoInfer is a utility type designed to block the automatic inference of types within the scope of a generic function.

Example:

```
// Automatic inference of types within the scope of a generic function.
function fn<T extends string>(x: T[], y: T) {
    return x.concat(y);
}
```

```
const r = fn(['a', 'b'], 'c'); // Type here is ("a" | "b" | "c")[]
```

With NoInfer:

```
// Example function that uses NoInfer to prevent type inference
function fn2<T extends string>(x: T[], y: NoInfer<T>) {
    return x.concat(y);
}

const r2 = fn2(['a', 'b'], 'c'); // Error: Type Argument of type '"c"' is
not assignable to parameter of type '"a" | "b"'.
```

Others

Errors and Exception Handling

TypeScript allows you to catch and handle errors using standard JavaScript error handling mechanisms:

Try-Catch-Finally Blocks:

```
try {
    // Code that might throw an error
} catch (error) {
    // Handle the error
} finally {
    // Code that always executes, finally is optional
}
```

You can also handle different types of error:

```
try {
    // Code that might throw different types of errors
} catch (error) {
    if (error instanceof TypeError) {
        // Handle TypeError
    } else if (error instanceof RangeError) {
        // Handle RangeError
    } else {
        // Handle other errors
    }
}
```

Custom Error Types:

It is possible to specify more specific error by extending on the Error `class`:

```
class CustomError extends Error {
    constructor(message: string) {
        super(message);
        this.name = 'CustomError';
    }
}

throw new CustomError('This is a custom error.');
```

Mixin classes

Mixin classes allow you to combine and compose behavior from multiple classes into a single class. They provide a way to reuse and extend functionality without the need for deep inheritance chains.

```
abstract class Identifiable {
    name: string = '';
    logId() {
        console.log('id:', this.name);
    }
}
abstract class Selectable {
    selected: boolean = false;
    select() {
        this.selected = true;
        console.log('Select');
    }
    deselect() {
        this.selected = false;
        console.log('Deselect');
    }
}
class MyClass {
    constructor() {}
}

// Extend MyClass to include the behavior of Identifiable and Selectable
interface MyClass extends Identifiable, Selectable {}

// Function to apply mixins to a class
function applyMixins(source: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            let descriptor = Object.getOwnPropertyDescriptor(
                baseCtor.prototype,
                name
            );
            if (descriptor) {
                Object.defineProperty(source.prototype, name, descriptor);
            }
        });
    });
}

// Apply the mixins to MyClass
applyMixins(MyClass, [Identifiable, Selectable]);
let o = new MyClass();
o.name = 'abc';
```

```
o.logId();
o.select();
```

Asynchronous Language Features

As TypeScript is a superset of JavaScript, it has built-in asynchronous language features of JavaScript as:

Promises:

Promises are a way to handle asynchronous operations and their results using methods like `.then()` and `.catch()` to handle success and error conditions.

To learn more:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Async/await:

Async/await keywords are a way to provide a more synchronous-looking syntax for working with Promises. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used within an async function to pause execution until a Promise is resolved or rejected.

To learn more:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

The following API are well supported in TypeScript:

Fetch API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Web Workers: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

Shared Workers: https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker

WebSocket: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Iterators and Generators

Both Interators and Generators are well supported in TypeScript.

Iterators are objects that implement the iterator protocol, providing a way to access elements of a collection or sequence one by one. It is a structure that contains a pointer to the next element in the iteration. They have a `next()` method that returns the next value in the sequence along with a boolean indicating if the sequence is `done`.

```
class NumberIterator implements Iterable<number> {
    private current: number;
```

```
    constructor(
        private start: number,
        private end: number
    ) {
        this.current = start;
    }

    public next(): IteratorResult<number> {
        if (this.current <= this.end) {
            const value = this.current;
            this.current++;
            return { value, done: false };
        } else {
            return { value: undefined, done: true };
        }
    }

    [Symbol.iterator](): Iterator<number> {
        return this;
    }
}

const iterator = new NumberIterator(1, 3);

for (const num of iterator) {
    console.log(num);
}
```

Generators are special functions defined using the `function*` syntax that simplifies the creation of iterators. They use the `yield` keyword to define the sequence of values and automatically pause and resume execution when values are requested.

Generators make it easier to create iterators and are especially useful for working with large or infinite sequences.

Example:

```
function* numberGenerator(start: number, end: number): Generator<number> {
    for (let i = start; i <= end; i++) {
        yield i;
    }
}

const generator = numberGenerator(1, 5);

for (const num of generator) {
    console.log(num);
}
```

TypeScript also supports async iterators and async Generators.

To learn more:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Iterator

TsDocs JSDoc Reference

When working with a JavaScript code base, it is possible to help TypeScript to infer the right Type by using JSDoc comments with additional annotation to provide type information.

Example:

```
/**
 * Computes the power of a given number
 * @constructor
 * @param {number} base – The base value of the expression
 * @param {number} exponent – The exponent value of the expression
 */
function power(base: number, exponent: number) {
    return Math.pow(base, exponent);
}
power(10, 2); // function power(base: number, exponent: number): number
```

Full documentation is provided to this link:
https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html

From version 3.7 it is possible to generate .d.ts type definitions from JavaScript JSDoc syntax. More information can be found here:
https://www.typescriptlang.org/docs/handbook/declaration-files/dts-from-js.html

@types

Packages under the @types organization are special package naming conventions used to provide type definitions for existing JavaScript libraries or modules. For instance using:

```
npm install --save-dev @types/lodash
```

Will install the type definitions of `lodash` in your current project.

To contribute to the type definitions of @types package, please submit a pull request to https://github.com/DefinitelyTyped/DefinitelyTyped.

JSX

JSX (JavaScript XML) is an extension to the JavaScript language syntax that allows you to write HTML-like code within your JavaScript or TypeScript files. It is commonly used in React to define the HTML structure.

TypeScript extends the capabilities of JSX by providing type checking and static analysis.

To use JSX you need to set the `jsx` compiler option in your `tsconfig.json` file. Two common configuration options:

- "preserve": emit .jsx files with the JSX unchanged. This option tells TypeScript to keep the JSX syntax as-is and not transform it during the compilation process. You can use this option if you have a separate tool, like Babel, that handles the transformation.
- "react": enables TypeScript's built-in JSX transformation. React.createElement will be used.

All options are available here: https://www.typescriptlang.org/tsconfig#jsx

ES6 Modules

TypeScript does support ES6 (ECMAScript 2015) and many subsequent versions. This means you can use ES6 syntax, such as arrow functions, template literals, classes, modules, destructuring, and more.

To enable ES6 features in your project, you can specify the `target` property in the tsconfig.json.

A configuration example:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "es6",
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist"
  },
  "include": ["src"]
}
```

ES7 Exponentiation Operator

The exponentiation (`**`) operator computes the value obtained by raising the first operand to the power of the second operand. It functions similarly to `Math.pow()`, but with the

added capability of accepting BigInts as operands. TypeScript fully supports this operator using as `target` in your tsconfig.json file `es2016` or larger version.

```
console.log(2 ** (2 ** 2)); // 16
```

The for-await-of Statement

This is a JavaScript feature fully supported in TypeScript which allows you to iterate over asynchronous iterable objects from target version es2018.

```
async function* asyncNumbers(): AsyncIterableIterator<number> {
    yield Promise.resolve(1);
    yield Promise.resolve(2);
    yield Promise.resolve(3);
}

(async () => {
    for await (const num of asyncNumbers()) {
        console.log(num);
    }
})();
```

New target meta-property

You can use in TypeScript the `new.target` meta-property which enables you to determine if a function or constructor was invoked using the new operator. It allows you to detect whether an object was created as a result of a constructor call.

```
class Parent {
    constructor() {
        console.log(new.target); // Logs the constructor function used to
create an instance
    }
}

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

const parentX = new Parent(); // [Function: Parent]
const child = new Child(); // [Function: Child]
```

Dynamic Import Expressions

It is possible to conditionally load modules or lazy load them on-demand using the ECMAScript proposal for dynamic import which is supported in TypeScript.

The syntax for dynamic import expressions in TypeScript is as follows:

```
async function renderWidget() {
    const container = document.getElementById('widget');
    if (container !== null) {
        const widget = await import('./widget'); // Dynamic import
        widget.render(container);
    }
}

renderWidget();
```

"tsc –watch"

This command starts a TypeScript compiler with `--watch` parameter, with the ability to automatically recompile TypeScript files whenever they are modified.

```
tsc --watch
```

Starting from TypeScript version 4.9, file monitoring primarily relies on file system events, automatically resorting to polling if an event-based watcher cannot be established.

Non-null Assertion Operator

The Non-null Assertion Operator (Postfix !) also called Definite Assignment Assertions is a TypeScript feature that allows you to assert that a variable or property is not null or undefined, even if TypeScript's static type analysis suggests that it might be. With this feature it is possible to remove any explicit checking.

```
type Person = {
    name: string;
};

const printName = (person?: Person) => {
    console.log(`Name is ${person!.name}`);
};
```

Defaulted declarations

Defaulted declarations are used when a variable or parameter is assigned a default value. This means that if no value is provided for that variable or parameter, the default value will be used instead.

```
function greet(name: string = 'Anonymous'): void {
    console.log(`Hello, ${name}!`);
}
greet(); // Hello, Anonymous!
greet('John'); // Hello, John!
```

Optional Chaining

The optional chaining operator `?.` works like the regular dot operator (`.`) for accessing properties or methods. However, it gracefully handles null or undefined values by terminating the expression and returning `undefined`, instead of throwing an error.

```
type Person = {
    name: string;
    age?: number;
    address?: {
        street?: string;
        city?: string;
    };
};

const person: Person = {
    name: 'John',
};

console.log(person.address?.city); // undefined
```

Nullish coalescing operator

The nullish coalescing operator `??` returns the right-hand side value if the left-hand side is `null` or `undefined`; otherwise, it returns the left-hand side value.

```
const foo = null ?? 'foo';
console.log(foo); // foo

const baz = 1 ?? 'baz';
const baz2 = 0 ?? 'baz';
console.log(baz); // 1
console.log(baz2); // 0
```

Template Literal Types

Template Literal Types allow to manipulate string value at type level and generate new string types based on existing ones. They are useful to create more expressive and precise types from string-based operations.

```
type Department = 'engineering' | 'hr';
type Language = 'english' | 'spanish';
type Id = `${Department}-${Language}-id`; // "engineering-english-id" |
"engineering-spanish-id" | "hr-english-id" | "hr-spanish-id"
```

Function overloading

Function overloading allows you to define multiple function signatures for the same function name, each with different parameter types and return type. When you call an overloaded function, TypeScript uses the provided arguments to determine the correct function signature:

```
function makeGreeting(name: string): string;
function makeGreeting(names: string[]): string[];

function makeGreeting(person: unknown): unknown {
    if (typeof person === 'string') {
        return `Hi ${person}!`;
    } else if (Array.isArray(person)) {
        return person.map(name => `Hi, ${name}!`);
    }
    throw new Error('Unable to greet');
}

makeGreeting('Simon');
makeGreeting(['Simone', 'John']);
```

Recursive Types

A Recursive Type is a type that can refer to itself. This is useful for defining data structures that have a hierarchical or recursive structure (potentially infinite nesting), such as linked lists, trees, and graphs.

```
type ListNode<T> = {
    data: T;
    next: ListNode<T> | undefined;
};
```

Recursive Conditional Types

It is possible to define complex type relationships using logic and recursion in TypeScript. Let's break it down in simple terms:

Conditional Types: allows you to define types based on boolean conditions:

```
type CheckNumber<T> = T extends number ? 'Number' : 'Not a number';
type A = CheckNumber<123>; // 'Number'
type B = CheckNumber<'abc'>; // 'Not a number'
```

Recursion: means a type definition that refers to itself within its own definition:

```
type Json = string | number | boolean | null | Json[] | { [key: string]:
Json };

const data: Json = {
    prop1: true,
    prop2: 'prop2',
    prop3: {
        prop4: [],
    },
};
```

Recursive Conditional Types combine both conditional logic and recursion. It means that a type definition can depend on itself through conditional logic, creating complex and flexible type relationships.

```
type Flatten<T> = T extends Array<infer U> ? Flatten<U> : T;

type NestedArray = [1, [2, [3, 4], 5], 6];
type FlattenedArray = Flatten<NestedArray>; // 2 | 3 | 4 | 5 | 1 | 6
```

ECMAScript Module Support in Node

Node.js added support for ECMAScript Modules starting from version 15.3.0, and TypeScript has had ECMAScript Module Support for Node.js since version 4.7. This support can be enabled by using the `module` property with the value `nodenext` in the tsconfig.json file. Here's an example:

```
{
  "compilerOptions": {
    "module": "nodenext",
    "outDir": "./lib",
    "declaration": true
  }
}
```

Node.js supports two file extensions for modules: `.mjs` for ES modules and `.cjs` for CommonJS modules. The equivalent file extensions in TypeScript are `.mts` for ES modules and `.cts` for CommonJS modules. When the TypeScript compiler transpiles these files to JavaScript, it will create `.mjs` and `.cjs` files.

If you want to use ES modules in your project, you can set the `type` property to "module" in your package.json file. This instructs Node.js to treat the project as an ES module project.

Additionally, TypeScript also supports type declarations in .d.ts files. These declaration files provide type information for libraries or modules written in TypeScript, allowing other developers to utilize them with TypeScript's type checking and auto-completion features.

Assertion Functions

In TypeScript, assertion functions are functions that indicate the verification of a specific condition based on their return value. In their simplest form, an assert function examines a provided predicate and raises an error when the predicate evaluates to false.

```
function isNumber(value: unknown): asserts value is number {
    if (typeof value !== 'number') {
        throw new Error('Not a number');
    }
}
```

Or can be declared as function expression:

```
type AssertIsNumber = (value: unknown) => asserts value is number;
const isNumber: AssertIsNumber = value => {
    if (typeof value !== 'number') {
        throw new Error('Not a number');
    }
};
```

Assertion functions share similarities with type guards. Type guards were initially introduced to perform runtime checks and ensure the type of a value within a specific scope. Specifically, a type guard is a function that evaluates a type predicate and returns a boolean value indicating whether the predicate is true or false. This differs slightly from assertion functions,where the intention is to throw an error rather than returning false when the predicate is not satisfied.

Example of type guard:

```
const isNumber = (value: unknown): value is number => typeof value ===
'number';
```

Variadic Tuple Types

Variadic Tuple Types are a features introduces in TypeScript version 4.0, let's start to learn them by revise what is a tuple:

A tuple type is an array which has a defined length, and were the type of each element is known:

```
type Student = [string, number];
const [name, age]: Student = ['Simone', 20];
```

The term "variadic" means indefinite arity (accept a variable number of arguments).

A variadic tuple is a tuple type which has all the property as before but the exact shape is not defined yet:

```
type Bar<T extends unknown[]> = [boolean, ...T, number];

type A = Bar<[boolean]>; // [boolean, boolean, number]
type B = Bar<['a', 'b']>; // [boolean, 'a', 'b', number]
type C = Bar<[]>; // [boolean, number]
```

In the previous code we can see that the tuple shape is defined by the `T` generic passed in.

Variadic tuples can accept multiple generics make them very flexible:

```
type Bar<T extends unknown[], G extends unknown[]> = [...T, boolean,
...G];

type A = Bar<[number], [string]>; // [number, boolean, string]
type B = Bar<['a', 'b'], [boolean]>; // ["a", "b", boolean, boolean]
```

With the new variadic tuples we can use:

- The spreads in tuple type syntax can now be generic, so we can represent higher-order operation on tuples and arrays even when we do not know the actual types we are operating over.
- The rest elements can occur anywhere in a tuple.

Example:

```
type Items = readonly unknown[];

function concat<T extends Items, U extends Items>(
    arr1: T,
    arr2: U
): [...T, ...U] {
    return [...arr1, ...arr2];
```

```
}

concat([1, 2, 3], ['4', '5', '6']); // [1, 2, 3, "4", "5", "6"]
```

Boxed types

Boxed types refer to the wrapper objects that are used to represent primitive types as objects. These wrapper objects provide additional functionality and methods that are not available directly on the primitive values.

When you access a method like `charAt` or `normalize` on a `string` primitive, JavaScript wraps it in a `String` object, calls the method, and then throws the object away.

Demonstration:

```
const originalNormalize = String.prototype.normalize;
String.prototype.normalize = function () {
    console.log(this, typeof this);
    return originalNormalize.call(this);
};
console.log('\u0041'.normalize());
```

TypeScript represents this differentiation by providing separate types for the primitives and their corresponding object wrappers:

- string => String
- number => Number
- boolean => Boolean
- symbol => Symbol
- bigint => BigInt

The boxed types are usually not needed. Avoid using boxed types and instead use type for the primitives, for instance `string` instead of `String`.

Covariance and Contravariance in TypeScript

Covariance and Contravariance are used to describe how relationships work when dealing with inheritance or assignment of types.

Covariance means that a type relationship preserves the direction of inheritance or assignment, so if a type A is a subtype of type B, then an array of type A is also considered a subtype of an array of type B. The important thing to note here is that the subtype

relationship is maintained this means that Covariance accept subtype but doesn't accept supertype.

Contravariance means that a type relationship reverses the direction of inheritance or assignment, so if a type A is a subtype of type B, then an array of type B is considered a subtype of an array of type A. The subtype relationship is reversed this means that Contravariance accept supertype but doesn't accept subtype.

Notes: Bivariance means accept both supertype & subtype.

Example: Let's say we have a space for all animals and a separate space just for dogs.

In Covariance, you can put all the dogs in the animals space because dogs are a type of animal. But you cannot put all the animals in the dog space because there might be other animals mixed in.

In Contravariance, you cannot put all the animals in the dogs space because the animals space might contain other animals as well. However, you can put all the dogs in the animal space because all dogs are also animals.

```
// Covariance example
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}

class Dog extends Animal {
    breed: string;
    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }
}

let animals: Animal[] = [];
let dogs: Dog[] = [];

// Covariance allows assigning subtype (Dog) array to supertype (Animal)
array
animals = dogs;
dogs = animals; // Invalid: Type 'Animal[]' is not assignable to type
'Dog[]'

// Contravariance example
type Feed<in T> = (animal: T) => void;

let feedAnimal: Feed<Animal> = (animal: Animal) => {
    console.log(`Animal name: ${animal.name}`);
```

```
};

let feedDog: Feed<Dog> = (dog: Dog) => {
    console.log(`Dog name: ${dog.name}, Breed: ${dog.breed}`);
};

// Contravariance allows assigning supertype (Animal) callback to subtype
(Dog) callback
feedDog = feedAnimal;
feedAnimal = feedDog; // Invalid: Type 'Feed<Dog>' is not assignable to
type 'Feed<Animal>'.
```

In TypeScript, type relationships for arrays are covariant, while type relationships for function parameters are contravariant. This means that TypeScript exhibits both covariance and contravariance, depending on the context.

Optional Variance Annotations for Type Parameters

As of TypeScript 4.7.0, we can use the `out` and `in` keywords to be specific about Variance annotation.

For Covariant, use the `out` keyword:

```
type AnimalCallback<out T> = () => T; // T is Covariant here
```

And for Contravariant, use the `in` keyword:

```
type AnimalCallback<in T> = (value: T) => void; // T is Contravariance
here
```

Template String Pattern Index Signatures

Template string pattern index signatures allow us to define flexible index signatures using template string patterns. This feature enables us to create objects that can be indexed with specific patterns of string keys, providing more control and specificity when accessing and manipulating properties.

TypeScript from version 4.4 allows index signatures for symbols and template string patterns.

```
const uniqueSymbol = Symbol('description');

type MyKeys = `key-${string}`;

type MyObject = {
    [uniqueSymbol]: string;
    [key: MyKeys]: number;
};
```

```
const obj: MyObject = {
    [uniqueSymbol]: 'Unique symbol key',
    'key-a': 123,
    'key-b': 456,
};

console.log(obj[uniqueSymbol]); // Unique symbol key
console.log(obj['key-a']); // 123
console.log(obj['key-b']); // 456
```

The satisfies Operator

The `satisfies` allows you to check if a given type satisfies a specific interface or condition. In other words, it ensures that a type has all the required properties and methods of a specific interface. It is a way to ensure a variable fits into a definition of a type Here is an example:

```
type Columns = 'name' | 'nickName' | 'attributes';

type User = Record<Columns, string | string[] | undefined>;

// Type Annotation using `User`
const user: User = {
    name: 'Simone',
    nickName: undefined,
    attributes: ['dev', 'admin'],
};

// In the following lines, TypeScript won't be able to infer properly
user.attributes?.map(console.log); // Property 'map' does not exist on
type 'string | string[]'. Property 'map' does not exist on type 'string'.
user.nickName; // string | string[] | undefined

// Type assertion using `as`
const user2 = {
    name: 'Simon',
    nickName: undefined,
    attributes: ['dev', 'admin'],
} as User;

// Here too, TypeScript won't be able to infer properly
user2.attributes?.map(console.log); // Property 'map' does not exist on
type 'string | string[]'. Property 'map' does not exist on type 'string'.
user2.nickName; // string | string[] | undefined

// Using `satisfies` operators we can properly infer the types now
const user3 = {
    name: 'Simon',
```

```
    nickName: undefined,
    attributes: ['dev', 'admin'],
} satisfies User;

user3.attributes?.map(console.log); // TypeScript infers correctly:
string[]
user3.nickName; // TypeScript infers correctly: undefined
```

Type-Only Imports and Export

Type-Only Imports and Export allows you to import or export types without importing or exporting the values or functions associated with those types. This can be useful for reducing the size of your bundle.

To use type-only imports, you can use the `import type` keyword.

TypeScript permits using both declaration and implementation file extensions (.ts, .mts, .cts, and .tsx) in type-only imports, regardless of `allowImportingTsExtensions` settings.

For example:

```
import type { House } from './house.ts';
```

The following are supported forms:

```
import type T from './mod';
import type { A, B } from './mod';
import type * as Types from './mod';
export type { T };
export type { T } from './mod';
```

using declaration and Explicit Resource Management

A `using` declaration is a block-scoped, immutable binding, similar to `const`, used for managing disposable resources. When initialized with a value, the `Symbol.dispose` method of that value is recorded and subsequently executed upon exiting the enclosing block scope.

This is based on ECMAScript's Resource Management feature, which is useful for performing essential cleanup tasks after object creation, such as closing connections, deleting files, and releasing memory.

Notes:

- Due to its recent introduction in TypeScript version 5.2, most runtimes lack native

support. You'll need polyfills for: `Symbol.dispose`, `Symbol.asyncDispose`, `DisposableStack`, `AsyncDisposableStack`, `SuppressedError`.

- Additionally, you will need to configure your tsconfig.json as follows:

```
{
    "compilerOptions": {
        "target": "es2022",
        "lib": ["es2022", "esnext.disposable", "dom"]
    }
}
```

Example:

```
//@ts-ignore
Symbol.dispose ??= Symbol('Symbol.dispose'); // Simple polify

const doWork = (): Disposable => {
    return {
        [Symbol.dispose]: () => {
            console.log('disposed');
        },
    };
};

console.log(1);

{
    using work = doWork(); // Resource is declared
    console.log(2);
} // Resource is disposed (e.g., `work[Symbol.dispose]()` is evaluated)

console.log(3);
```

The code will log:

```
1
2
disposed
3
```

A resource eligible for disposal must adhere to the `Disposable` interface:

```
// lib.esnext.disposable.d.ts
interface Disposable {
    [Symbol.dispose](): void;
}
```

The `using` declarations record resource disposal operations in a stack, ensuring they are

disposed in reverse order of declaration:

```
{
    using j = getA(),
        y = getB();
    using k = getC();
} // disposes `C`, then `B`, then `A`.
```

Resources are guaranteed to be disposed, even if subsequent code or exceptions occur. This may lead to disposal potentially throwing an exception, possibly suppressing another. To retain information on suppressed errors, a new native exception, `SuppressedError`, is introduced.

await using declaration

An `await using` declaration handles an asynchronously disposable resource. The value must have a `Symbol.asyncDispose` method, which will be awaited at the block's end.

```
async function doWorkAsync() {
    await using work = doWorkAsync(); // Resource is declared
} // Resource is disposed (e.g., `await work[Symbol.asyncDispose]()` is
evaluated)
```

For an asynchronously disposable resource, it must adhere to either the `Disposable` or `AsyncDisposable` interface:

```
// lib.esnext.disposable.d.ts
interface AsyncDisposable {
    [Symbol.asyncDispose](): Promise<void>;
}
```

```
//@ts-ignore
Symbol.asyncDispose ??= Symbol('Symbol.asyncDispose'); // Simple polify

class DatabaseConnection implements AsyncDisposable {
    // A method that is called when the object is disposed asynchronously
    [Symbol.asyncDispose]() {
        // Close the connection and return a promise
        return this.close();
    }

    async close() {
        console.log('Closing the connection...');
        await new Promise(resolve => setTimeout(resolve, 1000));
        console.log('Connection closed.');
    }
}

async function doWork() {
    // Create a new connection and dispose it asynchronously when it goes
```

```
out of scope
    await using connection = new DatabaseConnection(); //  Resource is
declared
    console.log('Doing some work...');
} // Resource is disposed (e.g., `await connection[Symbol.asyncDispose]()`
is evaluated)

doWork();
```

The code logs:

```
Doing some work...
Closing the connection...
Connection closed.
```

The `using` and `await using` declarations are allowed in Statements: `for`, `for-in`, `for-of`, `for-await-of`, `switch`.

Import Attributes

TypeScript 5.3's Import Attributes (labels for imports) tell the runtime how to handle modules (JSON, etc.). This improves security by ensuring clear imports and aligns with Content Security Policy (CSP) for safer resource loading. TypeScript ensures they are valid but lets the runtime handle their interpretation for specific module handling.

Example:

```
import config from './config.json' with { type: 'json' };
```

with dynamic import:

```
const config = import('./config.json', { with: { type: 'json' } });
```