TypeScript Class Notes

AltSchool Africa

Are you ready to learn TypeScript? Press `space` on your keyboard →

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

Read more about TypeScript?

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

JavaScript and More

TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor. A Result You Can Trust

Read more about TypeScript?

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

JavaScript and More

TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor. A Result You Can Trust

TypeScript code converts to JavaScript, which runs anywhere JavaScript runs: In a browser, on Node.js or Deno and in your apps. Safety at Scale

Read more about TypeScript?

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

JavaScript and More

TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor. A Result You Can Trust

TypeScript code converts to JavaScript, which runs anywhere JavaScript runs: In a browser, on Node.js or Deno and in your apps. Safety at Scale

TypeScript understands JavaScript and uses type inference to give you great tooling without additional code

Read more about TypeScript?

Table of Content

What are the things we will be covering?

Table of Content

What are the things we will be covering?

The Basics

- Static type-checking
- Non-exception Failures
- Types for Tooling
- tsc, the TypeScript compiler
- Emitting with Errors
- Explicit Types
- Erased Types
- Downleveling
- Strictness
- noImplicitAny
- strictNullChecks

TypeScript Compiler `tsc`

- The TypeScript compiler is a tool that takes TypeScript code and turns it into JavaScript code.
- The TypeScript compiler can be installed as a Node.js package.
- The TypeScript compiler can be run from the command line.
- The TypeScript compiler can be configured using a configuration file.
- The TypeScript compiler can be used to compile multiple files.
- The TypeScript compiler can be used to compile a project.

```
npm install -g typescript

tsc hello.ts

tsc --noEmitOnError hello.ts

tsc --init
```

tsconfig.json

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.

- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.
- Downleveling: You can use the target flag to specify the version of JavaScript that the TypeScript compiler should output. The default is ES3.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.
- Downleveling: You can use the target flag to specify the version of JavaScript that the TypeScript compiler should output. The default is ES3.
- Emitting with Errors: You can use the noEmitOnError flag to prevent the TypeScript compiler from emitting JavaScript code if there are any errors.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.
- Downleveling: You can use the target flag to specify the version of JavaScript that the TypeScript compiler should output. The default is ES3.
- Emitting with Errors: You can use the noEmitOnError flag to prevent the TypeScript compiler from emitting JavaScript code if there are any errors.
- Explicit Types: You can use the noImplicitAny flag to prevent TypeScript from inferring the any type.

tsconfig.json

- Erased Types: You can use the noUnusedLocals and noUnusedParameters flags to prevent TypeScript from emitting JavaScript code if there are any unused variables or parameters.

```
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "target": "ES5",
    "noEmitOnError": true
  }
}
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

```ts
let person: string | number = "helloTtypescript";

let result: number[] = person.split("T");
```

```
Type 'string[]' is not assignable to type 'number[]'.
  Type 'string' is not assignable to type 'number'.
```

```ts
//  //^?

console.log(result);
console.log("Hello", "AltSchool");
```

functions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

functions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

functions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

functions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

Running...

Peek into Generics

```
// Inside ./snippets/external.ts
export function emptyArray<T>(length: number) {
  return Array.from<T>({ length })
}
```

Put `emptyArray` function to work

Peek into Generics

```ts
// Inside ./snippets/external.ts
export function emptyArray<T>(length: number) {
  return Array.from<T>({ length })
}
```

Put `emptyArray` function to work

```
Running...
```

Peek into Generics

```
// Inside ./snippets/external.ts
export function emptyArray<T>(length: number) {
  return Array.from<T>({ length })
}
```

Put `emptyArray` function to work

Running...

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}
// Note that we didn't have to specify Type in this sample.
// The type was inferred - chosen automatically - by TypeScript.
let s1 = firstElement([1, 2, 4, 5])
let s2 = firstElement(['hello', 'dance'])
```

Solve this using TS Generics

Running...

```ts
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}

// longerArray is of type 'number[]'
const longerArray = longest([1, 2], [1, 2, 3]);
// longerString is of type 'alice' | 'bob'
const longerString = longest("alice", "bob");
// Error! Numbers don't have a 'length' property
const notOK = longest(10, 100);
```

```
Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.
```

Running...

Running...

Running...

Running...

Guideline to writing good generics

Guideline to writing good generics

- Push Type Parameters Down

Guideline to writing good generics

- Push Type Parameters Down
- Use Fewer Type Parameters

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
  arr: Type[],
  func: Func
): Type[] {
  return arr.filter(func);
}

const val = filter1([1, 2, 3, 4], n => n % 2 === 0)
const val2 = filter2([1, 2, 3, 4], n => n % 2 === 0)
```

Guideline to writing good generics

- Push Type Parameters Down
- Use Fewer Type Parameters

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
  arr: Type[],
  func: Func
): Type[] {
  return arr.filter(func);
}

const val = filter1([1, 2, 3, 4], n => n % 2 === 0)
const val2 = filter2([1, 2, 3, 4], n => n % 2 === 0)
```

- Type Parameters(Or any annotation used) Should Appear Twice

function overloading

```
function add3(a: number, b: number): number;
function add3(a: string, b: string): string;
function add3(a: any, b: any): any {
  return a + b;
}

add3("na", "me");
add3(99, 78);
let name2: any = "wale";
let age2: any = 99;
add3(name2, age2);
```

Enums

Running...

Type Manipulation

Type Manipulation

- keyof

Type Manipulation

- keyof
- typeof

Type Manipulation

- keyof
- typeof
- indexed access types

Type Manipulation

- keyof
- typeof
- indexed access types
- conditional types

Type Manipulation

- keyof
- typeof
- indexed access types
- conditional types
- mapped types

Type Manipulation

- keyof
- typeof
- indexed access types
- conditional types
- mapped types
- template-literal-types