Evolution Engineering

TypeScript Basic

Bootcamp [TypeScript]

Dmitry Kiselyov

eng.evolution.com

What is TypeScript?

- TypeScript is an open-source language which builds on JavaScript, one of the world's most used tools, by adding static type definitions.
- TypeScript validates that your code is working correctly.
- All valid JavaScript code is also TypeScript code (TS errors could be ignored).
- TypeScript code is transformed into JavaScript code via the TypeScript compiler or Babel.

Benefits:

- Safety & confidence
- Reliable refactoring
- Time & convenience & comfort

Types

JS	TS
null	null
undefined	undefined
boolean	boolean / Boolean
string	string / String
symbol	symbol / Symbol
number	number / Number
bigint	bigint
object	object / Object
function	Function
	any
	unknown
	Array / [] / Tuple
	void
	never
	enum

Don't ever use the types **Number, String, Boolean, Symbol, or Object** These types refer to non-primitive boxed objects that are almost never used appropriately in JavaScript code.

The type names Number, String, Boolean, Symbol and Object (starting with capital letters) are legal, but refer to some special built-in types that will very rarely appear in your code.

Always use number, string, boolean, symbol and object for types.

object VS Object

Types: *boolean, string, symbol, number, bigint*

```ts
let isEqual: boolean = true;
// let isEqual: boolean = 8; // error

isEqual = false;
// isEqual = 'hello'; // error



let odd: number = 1;
odd = 3;
// odd = {}; // error



let lyrics: string = 'I feel like an astronaut in the ocean';
let smallBigInteger: bigint = 11n;
let superSymbol: symbol = Symbol('super description');



// Automatically inferred type
let greeting = 'Hola';
greeting = 'Hallo';
// greeting = true; // error
```

Playground link

Types: *Function*

```ts
// NOTE: Do not ever use Function type
function invoke(callback: Function): void {
    const result = callback(1, 2, 3, true); // `any` everywhere

    return result;
}

// Type
type Subtract = (a: number, b: number) => number;
const subtract: Subtract = (a, b) => a - b;

// Interface
interface Multiply {
    (a: number, b: number): number;
}
const multiply: Multiply = (a, b) => a * b;

// Inferred type
const sum = (a: number, b: number): number => a + b;

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

// Optional parameter
function sumAll1(a: number, b: number, c?: number): number {
    return a + b + (c || 0);
}

function sumAll2(a: number, b: number, c: number = 0): number {
    return a + b + c;
}

// inferred type - c: number
function sumAll3(a: number, b: number, c = 0): number {
    return a + b + c;
}

// NOTE: Always specify return type for functions !!!
```

Playground link

Types: *Function warm-up*

Playground link

Types: *object*

```ts
// Type
type Legs = {
    amount: number;
    favorite: string;
};

type HomoSapiens = {
    name: string;
    surname?: string;
    isProgrammer: boolean;

    code: () => void;
    codeFast?: () => void;

    speak(): void;
    speakLoudly?(): void;

    hands: {
        amount: number;
        preferableHand: string;
    };

    legs: Legs;

    // recursion
    bestFriend: HomoSapiens;
    children: HomoSapiens[];
}
```

```ts
// Interface
interface Legs {
    amount: number;
    favorite: string;
}

interface HomoSapiens {
    name: string;
    surname?: string;
    isProgrammer: boolean;

    code: () => void;
    codeFast?: () => void;

    speak(): void;
    speakLoudly?(): void;

    hands: {
        amount: number;
        preferableHand: string;
    };

    legs: Legs;

    // recursion
    bestFriend: HomoSapiens;
    children: HomoSapiens[];
}
```

```ts
interface CommonObject {
    [key: string]: object;

    names: string[];
}
```

```ts
interface CommonObject {
    [key: number]: boolean;

    name: string;
    surname?: string;
}
```

```ts
// All object keys are string!
// Inheritance is important!

interface CommonObject {
    // a common type (including numeric key type)
    // e.g., function / Array / Date / RegExp / etc. is a subclass of Object)
    [key: string]: object;
    // a part of the common type (e.g., function is object - extends it)
    [key: number]: () => void;

    names: string[];
}
```

[Playground link]

Types: *null, undefined, void*

```ts
let undefinedVariable: undefined = undefined;
let nullVariable: null = null;
```

```ts
// strictNullChecks: false
// void = undefined | null

let empty1: void;
let empty2: void = undefined;
let empty3: void = null;
// let empty4: void = 1; // TS error

function printSum(a: number, b: number): void {
    console.log(a + b);

    // return; // TS is ok, presence doesn't matter
    // return null; // TS is ok
    // return undefined; // TS is ok
}
```

```ts
// strictNullChecks: true
// void = undefined ONLY

let empty1: void;
let empty2: void = undefined;
// let empty3: void = null; // TS error
// let empty4: void = 1; // TS error

function printSum1(a: number, b: number): void {
    console.log(a + b);

    // return; // TS is ok, presence doesn't matter
    // return null; // TS error
    // return undefined; // TS is ok
}
```

[Playground link (strictNullChecks: false)]

[Playground link (strictNullChecks: true)]

[Playground link (void VS undefined)]

```ts
// void VS undefined (strictNullChecks: true, void = undefined)

function getRandom(): number {
    return Math.random();
}

// `void` means you promise not to use the return value
// so it can be called with a callback that returns any value
function doIt1(callback: () => void): void {
    console.log(callback());
}
doIt1(getRandom); // TS is ok


function doIt2(callback: () => undefined): void {
    callback();
}
// doIt2(getRandom); // TS error
```

Types: *Array / []*

```ts
// one-dimensional array

// preferable style
const ids: string[] = [];
ids.push('id_1');
// ids.push(987654321); // TS error

const evens: Array<number> = []; // via generic type
evens.push(4);
// evens.push('even'); // TS error



// two-dimensional array

// preferable style
const numericMatrix: number[][] = [
    [11, 12, 13],
    [21, 22, 23],
    [31, 32, 33],
];

const stringMatrix: Array<Array<string>> = [
    ['11', '12', '13'],
    ['21', '22', '23'],
    ['31', '32', '33'],
];
```

Playground link

Types: *Tuple*

```ts
type Arguments1 = [string, number, ...number[]];
const args1: Arguments1 = ['*', 1, 2, 3];

type Arguments2 = [...string[], () => void];
const args2: Arguments2 = ['*', '+', () => {}];
// const args2: Arguments2 = [() => {}];

type Arguments3 = [boolean, ...string[], number];
const args3: Arguments3 = [true, '*', '+', 'hi', 777];

// waaaaaat?
type Arguments4 = [...string[]];

// RESTRICTIONS

// 1. Only one rest element per tuple
// type Arguments5 = [...boolean[], ...string[], number];

// 2.No optional elements after rest elements
// type Arguments6 = [...string[], number?];

type Tuple1 = [number, string];
// type Tuple1 = [number, string, ...null[]]; // DO NOT do this
type Tuple2 = [symbol, boolean];
// type Tuple2 = [symbol, boolean, ...undefined[]]; // DO NOT do this

type TupleFromTuples = [...Tuple1, ...Tuple2, ...Tuple1];
// type TupleFromTuples = [...Tuple1, ...Tuple2, ...Tuple1, ...string[]];
// type TupleFromTuples = [...string[], ...Tuple1, ...Tuple2, ...Tuple1];
```

Playground link

```ts
type Handler = (event: object) => void;
type HandlerDescription = [string, Handler, boolean?];
// NOTE: Optional parameter can be only in the end
// type HandlerDescription = [string?, Handler, boolean?]; // TS error

// Named Tuple:
// 1. Tuple members must all have names or all not have names
// 2. Question mark moves to the name
type NamedHandlerDescription = [eventType: string, handler: Handler, useCapture?: boolean];

const handlerDescription: HandlerDescription = [
    'click',
    () => alert('Clicked'),
    false, // useCapture
];

window.addEventListener(...handlerDescription);

```

Types: *warm-up*

Playground link

Types: *any, unknown*

```ts
let unknownVariable: unknown;
unknownVariable = 1;
unknownVariable = {};
unknownVariable = 987n;

// unknownVariable.prop1.prop2.sum(); // TS error

if (typeof unknownVariable === 'string') {
    // unknownVariable; // string

    unknownVariable.split('').reverse().join('');
} else if (typeof unknownVariable === 'object') {
    // unknownVariable; // object | null

    if (unknownVariable !== null) {
        // unknownVariable; // object
        unknownVariable.toString();
    }
} else {
    unknownVariable; // unknown
}
```

Playground link

```ts
let misteryVariable: any;
// misteryVariable = 1;
// misteryVariable = {};
// misteryVariable = 987n;

misteryVariable.prop1.prop2.sum(); // runtime error, TS doesn't work
misteryVariable(1, 2, true); // runtime error, TS doesn't work

// NOTE: `let` declaration without a type definition === `any`
// DO NOT ever write this way
let misteryVariable2;
misteryVariable2 = 'str';
misteryVariable2 = {};
misteryVariable2 = 1;
```

Types: *never*

```ts
// never: there is no a reachable endpoint
function throwError(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
    while (true) {
        // ...
    }
}
```

Playground link

Types: *enum*

```ts
// Numeric enums
// numeric enums are auto-incremented
enum Numeric {
    // Zero = 0,
    Zero,
    One,
    // OneAndHalf = 1.5,
    Two,
}

// String enums
// string enums allow you to give a meaningful and readable value when your code runs
// preferable style
enum CardinalPoint {
    North = 'north',
    South = 'south',
    East = 'east',
    West = 'west'
}

// Enum is type as well!
function getCoordinate(cardinalPoint: CardinalPoint): number {
    if (cardinalPoint === CardinalPoint.South || cardinalPoint === CardinalPoint.North) {
        // cardinalPoint; // CardinalPoint.South | CardinalPoint.North
        return 999;
    }

    // cardinalPoint; // CardinalPoint.East | CardinalPoint.West
    return 222;
}

getCoordinate(CardinalPoint.West);
```

Playground link

Types: *enum naming convention*

DOS:

✅ Use PascalCase for Enum **types** and **value** names.

✅ Use a singular name for Enum types.

DON'TS:

❌ Do not use an **Enum** suffix on Enum type names.

Playground link

```ts
// OK
enum CardinalPoint {
    // Enum values don't matter. This is your constants.
    // Write them using common sense and keeping consistency.
    North = 'north',
    South = 'SOUTH',
    West = 'West',
    SouthEast = 'south_east',
    SouthWest = 'SouthWest',
    NorthEast = 'north-east',
    NorthWest = 'northWest',
}

// NO - no `enum` suffix
enum CardinalPointEnum {
    // NO - PascalCase
    NORTH = 'north',
    // NO - PascalCase
    South_East = 'south_east',
    // NO - PascalCase
    southWest = 'south_west',
}

// NO - no plural
enum CardinalPoints {
    North = 'north',
    South = 'south',
}
```

Types: *Literal Types / as const*

Playground link

```ts
const TS_URL = 'https://www.typescriptlang.org/';
const PI = 3.14;

// readonly
interface ImmutableObject {
    readonly do: () => void;
    readonly id: number;
    readonly obj: {
        a: string;
        b: string;
    };
}

const immutableObject: ImmutableObject = {
    do: () => {},
    id: 3579,
    obj: {
        a: 'string 1',
        b: 'string 2',
    },
};
immutableObject.do();
// immutableObject.do = () => {}; // TS error
// immutableObject.id = 5; // TS error
// immutableObject.obj = {}; // TS error
immutableObject.obj.a = 'qwerty';

// Readonly
type ImmutableFromMutable = Readonly<{ property: string }>;
```

```ts
type ImmutableArray = readonly number[];
type ImmutableArrayTuple = readonly [number, string];

// Readonly
// type ImmutableArray = Readonly<number[]>;
// type ImmutableArrayTuple = Readonly<[number, string]>;

const immutableArray: ImmutableArray = [1, 2, 3];
// immutableArray.push(10);
// immutableArray[1] = 0;

interface ObjWithImmutableArrays {
    readonly numArr: number[];
    stringArr: readonly string[];
    readonly booleanArr: readonly boolean[];
}
const objWithImmutableArrays: ObjWithImmutableArrays = {
    numArr: [22, 23],
    stringArr: ['hi', 'hello'],
    booleanArr: [false, true],
};
// objWithImmutableArrays.numArr = [98, 97]; // TS error
objWithImmutableArrays.numArr.push(123);

objWithImmutableArrays.stringArr = ['bye'];
// objWithImmutableArrays.stringArr.push('hola'); // TS error

// objWithImmutableArrays.booleanArr = [true]; // TS error
// objWithImmutableArrays.booleanArr.push(false); // TS error
```

```ts
// `as const` works recursively

// const COORDS = [1, 2, 3, 4, 5]; // number[]
const COORDS = [1, 2, 3, 4, 5] as const;
// TS errors
// COORDS[2] = 10;
// COORDS.push(7);

const COORD_OBJECTS = [
    { x: 11, y: 12 },
    { x: 21, y: 22 },
] as const;
// TS errors
// COORD_OBJECTS.push({ x: 31, y: 32 });
// COORD_OBJECTS[0].x = 0;

const AGE_BY_NAME = {
    Name1: 11,
    Name2: 32,
    Groznyi: {
        title: 'King',
        age: Infinity,
    },
} as const;
// TS errors
// AGE_BY_NAME.Name1 = 11;
// AGE_BY_NAME.Groznyi = {};
// AGE_BY_NAME.Groznyi.title = 'Queen';
```

Types: *inferred array type*

```ts
// Tuple isn't inferred by default

const STRINGS = ['west', 'east']; // string[]
const STRINGS_NUMBERS = ['west', 'east', 450]; // (string | number)[]

function exists(
    array: number[][],
    value: number,
// ) {
): [
    repeatCount: number,
    found: boolean,
    coordinates: [column: number, row: number][],
] {
    let repeatCount = 0;
    const coordinates: [column: number, row: number][] = [];

    array.forEach((row, rowIndex) => {
        row.forEach((currentValue, columnIndex) => {
            if (currentValue === value) {
                repeatCount++;
                coordinates.push([rowIndex, columnIndex]);
            }
        });
    });

    return [repeatCount, !!repeatCount, coordinates];
    // NOTE: more preferable to define a return type
    // return [repeatCount, !!repeatCount, coordinates] as const;
}

const [repeatCount, found, coordinates] = exists([[1, 2, 3], [5, 7, 1], [10]], 1);
coordinates.forEach(() => {});
```

Playground link

Types: *Unions and Intersection Types*

```ts
interface Human {
    work(): void;
}

interface Animal {
    relax(): void;
}

// Union Types
type Id = string | number;
type Size = 'XS' | 'S' | 'M' | 'L' | 'XL';
type Odd = 1 | 3 | 5 | 7;



// Intersection Types
type Superman = Human & Animal;
const superman: Superman = {
    work() {},
    relax() {},
};
superman.relax();
superman.work();

type NewSuperman = Human & Animal & { relax(): string | number };
const newSuperman: NewSuperman = {
    work() {},
    // relax() {},
    relax() { return 7; },
};
```

Playground link

```ts
interface Human {
    work(): void;
}

interface Animal {
    relax(): void;
}

interface EnhancedSuperman extends Human, Animal {
    // relax(): string | number;
    code(): string;
}
const enhancedSuperman: EnhancedSuperman = {
    work() {},
    relax() {},
    // relax() { return 7; },
    code() { return 'coding...'; },
};
```

Types: *exhaustive check*

```ts
enum CardinalPoint {
  North = 'north',
  South = 'south',
  East = 'east',
  West = 'west'
}

function assertNever(arg: never): never {
  throw new Error(`Unexpected argument: ${arg}`);
}

function getCoordinate(cardinalPoint: CardinalPoint): [number, number] {
  switch (cardinalPoint) {
    case CardinalPoint.North:
      return [0, 0];
    case CardinalPoint.South:
      return [1, 1];
    case CardinalPoint.East:
      return [0, 1];
    case CardinalPoint.West:
      return [1, 0];

    default:
      return assertNever(cardinalPoint);
  }
}

function doAll(operation: '+' | '*', ...args: number[]): number {
    switch (operation) {
        case '+': return args.reduce((sum, value) => sum + value, 0);
        case '*': return args.reduce((product, value) => product * value, 1);

        default: assertNever(operation);
    }
}
```

Playground link

Types: *Type VS Interface*

```ts
// Type Alias
// A type alias is exactly that - a name for any type
// (objects, functions, primitives, unions, intersections, etc.)
type Value1 = string | number | boolean;
type Value2 = 'str' | 200 | true;
type Value3 = string;

type SomeFunction1 = {
    (index: number, code: string): string;
};
type SomeFunction2 = (index: number, code: string) => string;

type HomoSapiensType = {
    name: string;
};

type HomoSapiensClone = HomoSapiensType;

type SuperHomoSapiensType = HomoSapiensType & {
    isOk: boolean;
};

// Interface
// An interface declaration is another way to name an object type
// ONLY OBJECTS
interface SomeFunctionInterface1 {
    (index: number, code: string): string;
}

interface HomoSapiensInterface {
    name: string;
}

interface SuperHomoSapiensInterface extends HomoSapiensInterface {
    isOk: boolean;
}
```

Playground link

```ts
// Type Alias VS Interface

// Type aliases and interfaces are very similar,
// and in many cases you can choose between them freely.
// Almost all features of an interface are available in type,
// the key distinction is that a type cannot be re-opened to
// add new properties vs an interface which is always extendable.

// use `interface` until you need to use features from `type`

interface Country {
    name: string;
}

interface Country {
    size: number;
}

// TS errors
// type Country {
//     name: string;
// };

// type Country {
//     size: number;
// };

const Belarus: Country = { name: 'Belarus', size: 999_999 };
```

Types: *indexed access*

Playground link

```ts
interface Legs {
    amount: number;
    favorite: string;
}

interface HomoSapiens {
    name: string;
    surname?: string;
    isProgrammer: boolean;
    legs: Legs;

    move(x: string, y: string): string;
    speak(): void;

    bestFriend: HomoSapiens;
    children: HomoSapiens[];
}

type HomoSapiensName = HomoSapiens["name"];
type HomoSapiensSurname = HomoSapiens["surname"];
type HomoSapiensBestFriend = HomoSapiens["bestFriend"];

type HomoSapiensActions = HomoSapiens["move" | "speak"];

type SomeProperties = "legs" | "children";
// type SomeProperties = "legs" | "hands";
type SomeTypes = HomoSapiens[SomeProperties];
```

Types: *tips*

```ts
// How to find out the type?
// Inferred type VS Type definition

// redundant type definition
const regExp: RegExp = /test/;
const date: Date = new Date();
const divElement: HTMLDivElement = document.createElement('div');

// BETTER
// automatically inferred types (ONLY for `const`)
const regExp2 = /test/;
const date2 = new Date();
const divElement2 = document.createElement('div');



// doesn't work properly with `let`

// don't need define a type if specify a value immediately
let isTrue = true; // recognized as `boolean` automatically
// let isTrue: boolean = true; // the same, but type definition is redundant
isTrue = false;
// isTrue = 2; // TS error

// define a type if needed to specify a variaty of values
// let id = 1237321; // automatically recognized as `number` only
let id: number | string = 1237321; // automatically recognized as `number` only
id = 'unique_id';

// ALWAYS define a type if the value is not specified immediately
let newId; // recognized as `any` automatically
newId = 3;
newId = null;
newId = { id: 'id' };
newId = true;
```

Playground link

```ts
// Inferred type VS Type definition in function return type
// NO RETURN TYPE - why is it bad?

function concatStrings(a: string, b: string) {
    // if (!a || !b) {
    //     return;
    // }

    return `${a}_${b}`;
}

// more control of yourself and who will work with it later
// function concatStrings(a: string, b: string): string {
//     if (!a || !b) {
//         return;
//     }

//     return a + b;
// }

const result1 = concatStrings('Hi', 'Dimon!');
result1.toUpperCase();

const result2 = concatStrings('', '');
result2.toUpperCase();
```

Types: *warm-up*

Playground link

Classes

```ts
// private, protected, public & readonly & static
class TSClass {
    private readonly ID = "CLASS";
    // private title; // any
    private title: string;
    private count: number;

    static version = 'v0.1';
    // private static readonly version = 'v0.1'; // It's ok

    constructor(title: string, count: number) {
        this.title = title;
        this.count = count;
    }

    protected readonly getId = (): string => {
        return this.ID;
    }

    public doSmth(): string {
        return this.title.repeat(this.count);
    }
}

const tsClass = new TSClass('str', 789);
tsClass.doSmth();
// NOTE: JavaScript runtime constructs can still
// access a private or protected member.
// tsClass.title; // TS error, private property
// tsClass.getId(); // TS error, protected property
TSClass.version;
TSClass.version = 'v2'; // TS error, if it's `readonly`

// Parameter Properties: declaration in constructor
class TSClass2 {
    private timestamp: number;

    constructor(
        private title: string,
        private count: number,
        date: Date,
    ) {
        this.timestamp = date.valueOf();
    }
}
```

```ts
// extends
// Classes can only extend a single class

class TSClassBase {
    constructor(
        protected title: string,
        protected count: number,
    ) {}

    public doSmth(): string {
        return this.title.repeat(this.count);
    }

    // It's important that a derived class follow its base class contract.
    protected run(): number {
        return this.count++;
    }
}

// A derived class is always a subtype of its base class.
class TSClassExtended extends TSClassBase {
    constructor(
        title: string,
        count: number,
    ) {
        super(title, count);
    }

    public doSmthExtended(): string {
        return this.doSmth() + '_extended';
    }

    protected run(value: number = 0): number {
        return super.run() + value;
    }
}

const extendedClass = new TSClassExtended('str', 789);
extendedClass.doSmth();
extendedClass.doSmthExtended();
// extendedClass.title(); // TS error, protected property
```

[Playground link 1](#)

[Playground link 2](#)

```ts
// implements

// NO possibility to define `protected` or `private` properties or methods,
// because Interfaces define `public` contracts only,
// it doesn't make sense to have `protected` or `private` interfaces,
// which are related to implementation details.

// `implements` is only a check that the class can be treated as the interface type.
// It doesn't change the type of the class

interface TSClassInterface {
    doSmth(): string;
}
interface TSClassInterface2 {
    doSmth2: () => string;
}

class TSClass implements TSClassInterface, TSClassInterface2 {
    private timestamp: number;

    constructor(
        private title: string,
        private count: number,
        date: Date,
    ) {
        this.timestamp = date.valueOf();
    }

    public doSmth(): string {
        return this.title.repeat(this.count) + this.timestamp;
    }

    public doSmth2(): string {
        return this.title.repeat(this.count) + this.timestamp;
    }
}
```

Generic

```ts
// Generic function
function createArray<T>(value: T, length: number): T[] {
    return new Array(length).fill(value);
}
const arr1 = createArray(123, 10); // number[]
const arr2 = createArray('str', 10); // string[]
const arr3 = createArray(true, 10); // boolean[]
```

```ts
// make stronger types
function createArray<T extends number | string>(value: T, length: number): T[] {
    return new Array(length).fill(value);
}
const arr1 = createArray(123, 10); // number[]
const arr2 = createArray('str', 10); // string[]
// const arr3 = createArray(true, 10); // TS error
```

```ts
// Generic arrow function (works only with `extends`)
const createArray = <T extends number | string>(value: T, length: number): T[] => {
    return new Array(length).fill(value);
};
```

```ts
interface A {
    name: string;
}
interface B {
    name: string;
    surname: string;
}
interface C extends B {
    age: number;
}
type FunctionArgument = A | B | C;

function getName<T extends FunctionArgument>(arg: T): string {
    return arg.name.toUpperCase();
}
const getName1 = getName({ name: 'Dimon' });
const getName2 = getName({ name: 'Dimas', surname: 'K' });
const getName3 = getName({ name: 'Dimas', surname: 'K', age: 12 });
```

[Playground link]

```ts
// Generic type
interface ServerResponse<Key, Method> {
    key: Key;
    method: Method;
}
type ServerStringResponse = ServerResponse<string, 'GET' | 'POST'>;
const serverResponse: ServerStringResponse = {
    key: '123',
    // key: 123, // TS error
    method: 'POST',
};

interface ServerRequest<Key extends string | number = string> {
    key: Key;
}
type ServerStringRequest = ServerRequest;
// type ServerStringRequest = ServerRequest<string>; // the same
const serverRequest: ServerStringRequest = {
    key: '123',
    // key: 123, // TS error
};

type ServerNumberRequest = ServerRequest<number>;
// type ServerNumberRequest = ServerRequest<boolean>; // TS error
```

Generic Classes

```ts
class IDKeeper<Type> {
  constructor(
    public id: Type,
  ) {}

  public setNewId(id: Type): void {
    this.id = id;
  }
}

const stringIdKeeper = new IDKeeper("secret key");
stringIdKeeper.id; // string
stringIdKeeper.setNewId; // (id: string) => void

const numberIdKeeper = new IDKeeper(937_99_92);
numberIdKeeper.id; // number
numberIdKeeper.setNewId; // (id: number) => void
```

Playground link

```ts
class IDKeeper<Type extends string | number = string> {
  constructor(
    public id?: Type,
  ) {}

  public setNewId(id: Type): void {
    this.id = id;
  }
}

const stringIdKeeper = new IDKeeper("secret key"); // id: string
const numberIdKeeper = new IDKeeper(937_99_92); // id: number
// const booleanIdKeeper = new IDKeeper(true); // TS error

const idKeeperDefault = new IDKeeper(); // id: string - default type
const idKeeper = new IDKeeper<number>(); // id: number
```

TypeScript + React

```tsx
import React from 'react';

interface TitleProps {
    subtitleText: string;
}

interface TitleState {
    clickCount: number;
}

class Title extends React.Component<TitleProps, TitleState> {
    state = {
        clickCount: 0,
    };

    private onClick: React.MouseEventHandler<HTMLSpanElement> = (event) => {
        this.setState({ clickCount: this.state.clickCount + 1 });
        alert('Welcome');
    };

    render() {
        const { subtitleText } = this.props;
        const { clickCount } = this.state;
        const subtitle = <span>{subtitleText}</span>; // JSX.Element

        return (
            <div>
                <div onClick={this.onClick}>
                    TypeScript Bootcamp
                </div>
                <div>
                    Clicked: {clickCount}
                </div>
                {subtitle}
            </div>
        );
    }
}
```

[Playground link]

[Codesandbox link]

[CRA with TS]

```tsx
// Stateless Component (deprecated)
interface TitleTextProps {
    text: string;
    isVisible?: boolean;
}

function TitleText({ text, children, isVisible }: React.PropsWithChildren<TitleTextProps>): React.ReactNode {
    if (isVisible) {
        return null;
    }

    return <span>{text}{children}</span>;
}

const SmallTitleText: React.~~StatelessComponent~~<TitleTextProps> = ({ text, children, isVisible }) => {
    if (isVisible) {
        return null;
    }

    return <span>{text} - {children}</span>;
}
```

Thank you!