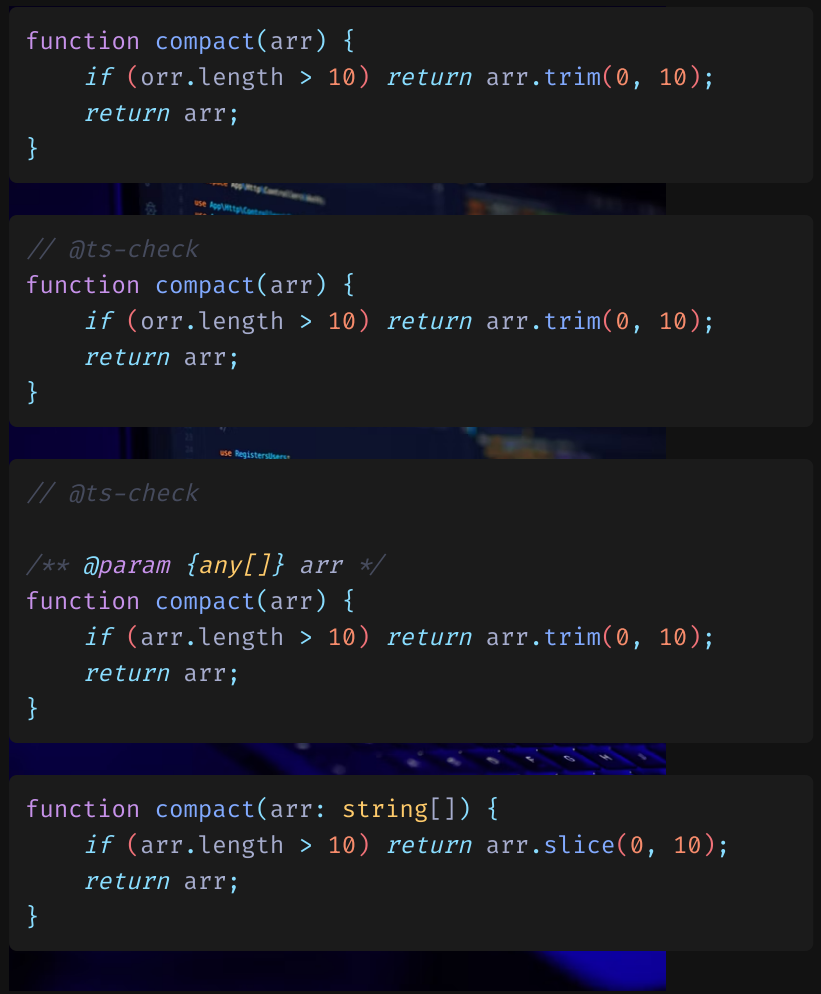
A short journey from TypeScript to JavaScript

TypeScript: optional static typing for JavaScript



What is TypeScript?

> TypeScript is a typed superset of JavaScript [...]

Or

> TypeScript is JavaScript with syntax for types.

- ## 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.

Why TypeScript?

- Early spotted bugs
- type-annotations
- Readability

> Reserve a space for static type syntax inside the ECMAScript language.

- Rich IDE support
- stateofjs
- Safer refactoring

> Back when the first State of JS survey took place, only 21% of you used TypeScript compared to 69% today.

- Type inference
- Availability of new and future JavaScript
- stackoverflow
- features

What are types?

Type Annotations

```
let age: number;
age = 18;

const isTruthy = !!1;

function equals(x: number, y: number): boolean {
    return x === y;
}

function compose<T extends Function>(...funcs: Function[]): T {
    return funcs.reverse().reduce((a, b) => (...args) => b(a(...args)));
}
```

```
function printPoint(point: { x: number; y: number; z?: number }) {
    console.table(point);
}
function printAnyhow(anyhow: Record<string | number | symbol, unknown>) {
    console.table(anyhow);
}
```

Types Overview

- null
- undefined
- boolean
- bigint
- number
- string
- symbol

- any
- unknown
- never
- void(a function which returns `` `undefined` `` or has no return value)
- interface
- type

Unions

```
function isTruthy(value: true | false) {}
function isActive(value: 'active' | 'inactive' | 'idle') {}
```

```
type MissingNo = 404;
type Location = {
    x: number;
    y: number;
}
type Data = [
    location: Location,
    timestamp: string,
]
type Size = "small" | "medium" | "large";
```

```
type Location = { x: number } & { y: number }
type Response = { data: {} };
type Data = Response['data'];
```

```
const data = {};
const Data = typeof Data;
```

```
const createFixtures = () => ({});
type Fixtures = ReturnType<typeof createFixtures>;
```

```
type Artist = { name: string; bio: string; }
type Subscriber<Type> = {
    [Property in keyof Type]: (newValue: Type[Property])
}
type ArtistSub = Subscriber<Artist>;
```

```
type HasFourLegs<Animal> = Animal extends { legs: 4 } ? A
type Animals = Bird | Dog | Ant | Wolf;
type FourLegs = HasFourLegs<Animal>;
```

```
type SupportedLangs = 'en' | 'pt' | 'zh';
type FooterLocaleIDs = 'header' | 'footer';

type AllLocaleIDs = `${SupportedLangs}_${FooterLocaleIDs}
```

Interfaces

```
interface JSONResponse extends Response, HTTPAble {
    version: number;
    payloadSize: number;

    update: (retryTimes: number) => void;
    update(retryTimes: number): void;

    (): JSONResponse;

    new(s: string): JSONResponse;

    [key: string]: number;
    readonly body: string;
}

interface APICall<Response extends { status: number }> {
    data: Response;
}

interface Expect {
    (matcher: boolean): string;
    (matcher: string): boolean;
}
```

Interface

```
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}

const bear = getBear()
bear.name
bear.honey
```

Type

```
type Animal = {
  name: string
}

type Bear = Animal & {
  honey: boolean
}

const bear = getBear();
bear.name;
bear.honey;
```

Type Assertions

```
const canvasEle = document.getElementById('a-canvas') as HTMLCanvasElement;
const handleCellTouched = (e: MouseEvent) => {
    const cell = e.target as HTMLTableCellElement;
}
const a = 1 as unknown as string;
const radioEleList = document.querySelectorAll<HTMLSelectElement>('input[type=radio]');

function getName(user: { name?: string }): string {
    return user.name!;
}
```

Literal Types

```ts
const PLATFORM = 'LINUX';
let name: 'Lily' = 'Lily';
// name = 'Lisa'; // ERROR

type LiteralTypes = 'string' | 100 | false;
const aList = ['A', 'B', 'C'] as const;
```

Enums

Enums are a feature added to JavaScript by TypeScript which allows for describing a value which could be one of a set of possible named constants. Unlike most TypeScript features, this is not a type-level addition to JavaScript but something added to the language and runtime. Because of this, it's a feature which you should know exists, but maybe hold off on using unless you are sure.

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

```
"use strict";
var Direction;
(function (Direction) {
    Direction[Direction["Up"] = 1] = "Up";
    Direction[Direction["Down"] = 2] = "Down";
```

Generics

```
type StringArray = Array<string>;
type ObjectWithNameArray = Array<{ name: string }>;
function identify<T> (arg: T) {
    return arg;
}
interface HttpResponse<T = unknown> {
    status: number;
    data: {
        code: number;
        message: string;
        result?: T
    };
}
interface GenericIdentityFn {
    <T>(arg: T): T;
}
class DragableObject { }
class Component<T extends Component> {
    draw(arg: T) {
        // ...
    }
}
const getProperty = <T, K extends keyof T>(obj: T, key: K) => obj[key];
```

Structural Type (Duck Typing)

```ts
interface Point {
    x: number;
    y: number;
}

class VirtualPoint {
    constructor(public x: number, public y: number) {}
}

function printPoint(point: Point) {
    console.table(point);
}

const point1 = { x: 1, y: 1 };
const point2 = new VirtualPoint(1, 1);
printPoint(point1);
printPoint(point2);

type Coordinate1 = { x: number; y: number };
type Coordinate2 = { x: number; y: number; z: number };
const printCoordinate = <T extends Coordinate1>(v: T) => console.log(v);
printCoordinate({ x: 1, y: 2 });
printCoordinate({ x: 1, y: 2, z: 3 });
```

Type Compatibility

```
interface Pet {
    name: string;
}
class Dog {
    name!: string;
}
let dog: Pet = new Dog;
```

```
interface Pet {
    name: string;
}
let dog = { name: 'Julia', owner: 'Michael' }

function greet(pet: Pet) {/*  ...  */}
greet(dog);
```

```
let x = (a: number) ⇒ 0;
let y = (a: number, b: number) ⇒ 0;

y = x; // Yep
x = y; // Nope
```

```
let x = () ⇒ ({ x: number, y: number });
let y = () ⇒ ({ x: number, y: number, z: number });
```

Class Members

```
class Customer {}

class Pallet {}

class Greeter {
    readonly name = 'Lily';

    #id = 0
    // private id = 0;

    // Overloads
    constructor(customer: Custom);
    constructor(pallet: Pallet) {}

    get id() {
        return this.#id;
    }

    set id(value) {
        this.#id = value
    }

    sayHi() {
        console.log('Hi, ' + this.name);
```

Class Heritage

```
interface Pingable {}
interface Pongable {}
class Radar {}
class Sonar extends Radar implements Pingable, Pongable {}
```

Member Visibility

```
class Account {}
class User extends Account {
    #attributes: Map<any, any>;
    name!: string;
    roles = ['user'];
    readonly createAt = Date.now();

    constructor(public id: string, public email: string) {
        super(id);
    }

    verifyName = (name: string) => {}

    sync(): Promise<boolean>;
    sync(cb: ((res: string) => void)): void;
    sync(cb?: ((res: string) => void)): void | Promise<boolean> {
        // ...
    }

    public greet() {
        console.log(`Hi, ${this.name}, I'm a ${Greeter.alias}!`);
    }

    private makeRequest() {}
```

Abstract classes & members

```
abstract class Base {
    abstract getName(): string;

    getAge() {
        return 18;
    }
}
class Derived extends Base {
    getName() {
        return 'Sid';
    }
}
function greet(ctor: new () => Base) {
    const instance = new ctor();
    instance.getName();
}
// greet(Base);
greet(Derived);
```

Decorators and Attributes

```
import {
    Syncable,
    triggersSync,
    preferCache,
    required
} from 'mylib';
@Syncable
class User {
    @triggersSync()
    save() {
        // ...
    }

    @preferCache(false)
    get displayName() {
        // ...
    }

    update(@required info: Partial<User>) {
        // ...
    }
}
```

Narrowing

```ts
function handleInput(input: string | number | boolean | sym
    if (typeof input === 'string') {/** ... */}
    else if (input instanceof Date) {/** ... */}
    else if ('id' in input) {/** ... */}
    else if (Array.isArray(input)) {/** ... */}
}

type Response =
    | { statis: 200, data: any }
    | { status: 301, to: string }
    | { status: 400, error: Error }

function isElement(node: Node): node is Element {
    return node instanceof Element;
}
function assertElement(obj: any): asserts obj is Element {
    if (!(obj instanceof Element)) {
        throw new Error("Not a Element");
    }
}
```

```ts
const data1 = {
    name: 'Misky'
};
const data2 = {
    name: 'Misky'
} as const;
```

```ts
type Shape = Circle | Square;

function getArea(shape: Shape) {
    switch (shape.kind) {
        case "circle":
            return Math.PI * shape.radius ** 2;
        case "square":
            return shape.sideLength ** 2;
        default:
            const _exhaustiveCheck: never = shape;
            return _exhaustiveCheck;
    }
}
```

Utility Types

How to study?

Resources for further learning

- TypeScript Documentation
- Type Challenge
- Books
  - TypeScript Deep Dive
  - 深入理解 TypeScript
- Videos
  - Learn TypeScript From Scratch!
- 知乎专题-TypeSript的另一面：类型编程