TypeScript – Advanced Types I

Pavel Tobiáš

FlowUp

2020–2021

Overview

1. Typing Objects
   - Structural Typing
   - Interfaces vs. Type Aliases
2. Combined Types
   - Union Types
   - Intersection Types
3. Type Discrimination
   - Type Discriminant Pattern

Typing Objects

Structural Typing

- Types with the same properties are considered equivalent.

```typescript
interface Person {
    age: number;
    name: string;
}

interface Dog {
    age: number;
    name: string;
}

let joe: Person = { name: 'Joe', age: 18 };
let rex: Dog = { name: 'Rex', age: 5 };

joe = rex; // Valid.
rex = joe; // Also valid.
```

Structural Typing (OOP Example)

```
interface Weapon {
    raise(): void;
}

interface Complaint {
    raise(): void;
}

class Sword implements Weapon {
    raise(): void { }
}

class Test {
    c: Complaint = new Sword();
}
```

- This is valid TypeScript, but would **not** be valid in languages with nominal (as opposed to structural) type systems.

Structural Subtypes (1)

- Object type `S` is a subtype of (and therefore assignable to) object type `T` if every property of `T` is also present in `S`.

```
interface Employee {
    name: string;
    monthlyWage: number;
}

interface Programmer extends Employee {
    knownLanguages: string[];
}

interface Tester {
    name: string;
    monthlyWage: number;
    canWriteUnitTests: boolean;
}
```

Structural Subtypes (2)

```
let programmer: Programmer = {
    name: 'Bob␣Bracket',
    monthlyWage: 50000,
    knownLanguages: ['Python', 'Rust'],
};

let tester: Tester = {
    name: 'Alice␣Asserti',
    monthlyWage: 10000,
    canWriteUnitTests: true,
};

let employee: Employee;
employee = programmer; // Works.
employee = tester; // Also works.
programmer = employee; // Does NOT work.
tester = employee; // NEITHER does this.
```

Structural Subtypes (3)

- More generally, object type `S` is a subtype of object type `T` if it holds for every property of `T` that `S` has a property of the same name and a type that is a **subtype** of `T`'s corresponding property.

```
interface Company {
    employees: Employee[];
}

interface Startup {
    employees: Programmer[];
}

let company: Company = { employees: [] };
let startup: Startup = { employees: [] };
company = startup; // Works.
startup = company; // Does NOT work.
```

- See this example at TypeScript Playground.

Primitive Subtypes

- A primitive type `S` is a subtype of a primitive type `T` if every value assignable to `S` is also assignable to `T`.

```
let a: string = 'foo';
let b: 'foo' = 'foo';
a = b; // Valid.
b = a; // NOT valid.

interface UserV1 { isAdmin?: boolean }
interface UserV2 { isAdmin?: true }
let admin1: UserV1 = { isAdmin: true };
let admin2: UserV2 = { isAdmin: true };
admin1 = admin2; // Valid.
admin2 = admin1; // NOT valid.
```

- See this example at TypeScript Playground.

Interfaces vs. Type Aliases

- Interfaces can be redefined. (Definitions are merged together.)

```
interface I { foo: string }
interface I { bar: number }
const value: I = { foo: '', bar: 0 };

type T = { foo: string };
type T = { bar: number };
```

- Type aliases support mapped object types.
- Type aliases can label **any** type expression (not just object types).
- **Both** can be `implement`'ed by a class.

  (A class can only implement an object type or intersection of object types with statically known members.)

- Should one **ever** use an interface? (Further reading.)

Combined Types

Union Types

- A value is assignable to the type `T`$_1$ `|` `T`$_2$ `|` `...` `|` `T`$_n$ if it is assignable to **at least one** of the types `T`$_1$, `T`$_2$, `...`, `T`$_n$.

```
type Bird = { name: string; flying: boolean };
type Cat = { name: string; lives: number };
type Animal = Bird | Cat;

declare const bird: Bird;
declare const cat: Cat;
declare const animal: Animal;

const a: Animal = bird; // Valid.
const b: Bird = animal; // NOT valid; could be a cat.
const c: Cat = animal; // NOT valid; could be a bird.
const d = animal.name; // Valid; both have names.
const e = animal.flying; // NOT valid; could be a cat.
const f = animal.lives; // NOT valid; could be a bird.
```

Union Types – Example

```typescript
type Employee = { hourlyPay: number, id: number };
type Contractor = { hourlyPay: number, contract: Blob };
type Person = Employee | Contractor;

// Checks the average hourly pay among team members who
// might be either regular employees or contractors.
function averageHourlyPay(people: Person[]): number {
    const paySum = people.reduce(
        (acc, person) => acc + person.hourlyPay,
        0
    );
    return paySum / people.length;
}
```

Intersection Types

- A value is assignable to the type `T`$_1$ `&` `T`$_2$ `& ... &` `T`$_n$ if it is assignable to **all** of the types `T`$_1$, `T`$_2$, `...`, `T`$_n$.

```
type Bird = { name: string; flying: boolean };
type Cat = { name: string; lives: number };
type Gryphon = Bird & Cat;

declare const bird: Bird;
declare const cat: Cat;
declare const gryphon: Gryphon;

const a: Gryphon = bird; // NOT valid (only a subset).
const b: Gryphon = cat; // NOT valid (only a subset).
const c: Bird = gryphon; // Valid (superset).
const d: Cat = gryphon; // Valid (superset).
const e = gryphon.name; // Valid (both have it).
const f = gryphon.flying; // Valid (at least 1 has it).
const g = gryphon.lives; // Valid (at least 1 has it).
```

Intersection Types – Example

```typescript
type Professor = { taughtSubjects: string[] };
type Student = { registeredSubjects: string[] };

// Checks whether a university IS user is teaching a
// subject that they are also studying (undesirable).
// The user must be a professor and a student at the
// same time for this check to make sense.
function studiesOwnSubject(
    user: Professor & Student
): boolean {
    return user.registeredSubjects
        .some(s => user.taughtSubjects.includes(s));
}
```

Type Discrimination

Motivation

```
type UserAccount = { userId: number };
type CreditCard = { cardNumber: string, ccv: string };

function pay(info: UserAccount | CreditCard): void {
    if ((info as UserAccount).userId != null) {
        // Need for repeated type-casting.
        console.log((info as UserAccount).userId);

        // This IS but SHOULD NOT be ok in this branch!
        console.log((info as CreditCard).cardNumber);
    }
}
```

- But wait! There is a better way!

Property Presence Check

```
type UserAccount = { userId: number };
type CreditCard = { cardNumber: string, ccv: string };

function pay(info: UserAccount | CreditCard): void {
    if ('userId' in info) {
        console.log(info.userId);
    } else {
        // 'userId' not in info => must be CreditCard.
        console.log(info.cardNumber);
    }
}
```

Variable Value Check

```
type UserAccount = { userId: number };
type CreditCard = { userId: null, cardNumber: string };

function pay(info: UserAccount | CreditCard): void {
    // Here, `userId: number | null`.
    if (info.userId != null) {
        // `userId: null' must be FALSE
        // => `userId: number' => `info: UserAccount`.
        console.log(info.userId);
    } else {
        // `userId == null'
        // => `userId: number' must be FALSE
        // => `userId: null' => `info: CreditCard'.
        console.log(info.cardNumber);
    }
}
```

Variable Type Check

```
type CreditCard = { cardNumber: string, ccv: string };

// Accepts user id or credit card info.
function pay(info: number | CreditCard): void {
    if (typeof info === 'number') {
        // `info: number`.
        console.log(info);
    } else {
        // `info: number` is FALSE => `info: CreditCard`.
        console.log(info.cardNumber);
    }
}
```

Type Discriminant Pattern

Step-by-step examples at the TypeScript Playground:

1. "Natural" duck-typing.
2. Usage of type discriminant.
3. Common properties.
4. (The same thing using interfaces.)