Understanding TypeScript’s type notation

2ality.com/2018/04/type-notation-typescript.html

What you’ll learn

After reading this post, you should be able to understand what the following code means:

```
interface Array<T> {
  concat(...items: Array<T[] | T>): T[];
  reduce<U>(
    callback: (state: U, element: T, index: number, array: T[]) => U,
    firstState?: U
  ): U;
  // ···
}
```

If you think this is cryptic – then I agree with you. But (as I hope to prove) this notation is relatively easy to learn. And once you understand it, it gives you immediate, precise and comprehensive summaries of how code behaves. No need to read long descriptions in English.

Trying out the code examples

TypeScript has an online playground. In order to get the most comprehensive checks, you should switch on everything in the “Options” menu. This is equivalent to running the TypeScript compiler in `--strict` mode.

Specifying the comprehensiveness of type checking

I recommend to always use TypeScript with the most comprehensive setting, `--strict`. Without it, programs are slightly easier to write, but we also lose many benefits of static type checking. Currently, this setting switches on the following sub-settings:

- `--noImplicitAny`: If TypeScript can’t infer a type, we must specify it. This mainly applies to parameters of functions and methods: With this settings, we must annotate them.
- `--noImplicitThis`: Complain if the type of `this` isn’t clear.
- `--alwaysStrict`: Use JavaScript’s strict mode whenever possible.
- `--strictNullChecks`: `null` is not part of any type (other than its own type, `null`) and must be explicitly mentioned if it is a acceptable value.
- `--strictFunctionTypes`: stronger checks for function types.
- `--strictPropertyInitialization`: If a property can’t have the value `undefined`, then it must be initialized in the constructor.

More info: chapter “Compiler Options” in the TypeScript Handbook.

Types

In this chapter, a type is simply a set of values. The JavaScript language (not TypeScript!) has only eight types:

1. Undefined: the set with the only element `undefined`
2. Null: the set with the only element `null`
3. Boolean: the set with the two elements `false` and `true`
4. Number: the set of all numbers
5. BigInt: the set of all arbitrary-precision integers
6. String: the set of all strings
7. Symbol: the set of all symbols
8. Object: the set of all objects (which includes functions and arrays)

All of these types are *dynamic*: we can use them at runtime.

TypeScript brings an additional layer to JavaScript: *static types*. These only exist when compiling or type-checking source code. Each storage location (variable, property, etc.) has a static type that predicts its dynamic values. Type checking ensures that these predictions come true. And there is a lot that can be checked *statically* (without running the code). If, for example the parameter `x` of a function `f(x)` has the static type `number`, then the function call `f('abc')` is illegal, because the parameter `'abc'` has the wrong static type.

Type annotations

A colon after a variable name starts a *type annotation*: the *type expression* after the colon describes what values the variable can have. For example, the following line tells TypeScript that `x` will only ever store numbers:

```
let x: number;
```

You may wonder if `x` being initialized with `undefined` doesn't violate the static type. TypeScript gets around this problem by not letting us read `x` before we assign a value to it.

Type inference

Even though every storage location has a static type in TypeScript, we don't always have to explicitly specify it. TypeScript can often infer it. For example, if we write:

```
// %inferred-type: number
let myNumber = 123;
```

Then TypeScript infers that `myNumber` has the static type `number`.

Specifying types via type expressions

The type expressions after the colons of type annotations range from simple to complex and are created as follows.

Basic types are valid type expressions:

- Static types for JavaScript's dynamic types:
  - `undefined`, `null`
  - `boolean`, `number`, `string`
  - `symbol`
  - `object`.
- TypeScript-specific types:
  - `Array` (not technically a type in JS)
  - `any` (the type of all values)
  - Etc.

Note that "`undefined` as a value" and "`undefined` as a type" are both written as `undefined`. Depending on where we use it, it is interpreted as a value or as a type. The same is true for `null`.

There are many ways of combining basic types to produce new, *compound types*. For example, via *type operators* that combine types similarly to how the set operators *union* (`∪`) and intersection (`∩`) combine sets.

More on this topic soon.

Type aliases

With `type` we can create a new name (an alias) for an existing type:

```
type Age = number;
const age: Age = 82;
```

Typing Arrays

Arrays are used in the following two roles in JavaScript (and sometimes a mix of the two):

- Lists: All elements have the same type. The length of the Array varies.
- Tuple: The length of the Array is fixed. The elements do not necessarily have the same type.

Arrays as lists

There are two ways to express the fact that the Array `arr` is used as a list whose elements are all numbers:

```
let arr: number[] = [];
let arr: Array<number> = [];
```

Normally, TypeScript can infer the type of a variable if there is an assignment. In this case, we actually have to help it, because with an empty Array, it can't determine the type of the elements.

We'll get back to the angle brackets notation (`Array<number>`) later.

Arrays as tuples

If we store a two-dimensional point in an Array then we are using that Array as a tuple. That looks as follows:

```
let point: [number, number] = [7, 5];
```

The type annotation is needed for Arrays-as-tuples because TypeScript infers list types, not tuple types:

```
// %inferred-type: number[]
let point = [7, 5];
```

Another example for tuples is the result of `Object.entries(obj)` : an Array with one [key, value] pair for each property of `obj` .

```
// %inferred-type: [string, number][]
const entries = Object.entries({ a: 1, b: 2 });

assert.deepEqual(
  entries,
  [[ 'a', 1 ], [ 'b', 2 ]]);
```

The inferred type is an Array of tuples.

Function types

This is an example of a function type:

```
(num: number) => string
```

This type comprises all functions that accept a single parameter, a number, and return a string. Let’s use this type in a type annotation:

```
const func: (num: number) => string =
  (num: number) => String(num);
```

Again, we don’t need a type annotation here because TypeScript is good at inferring function types:

```
// %inferred-type: (num: number) => string
const func = (num: number) => String(num);
```

A more complicated example

The following example is more complicated:

```
function stringify123(callback: (num: number) => string) {
  return callback(123);
}
```

We are using a function type to describe the parameter `callback` of `stringify123()` . Due to this type annotation, TypeScript rejects the following function call.

```
// @ts-ignore: Argument of type 'NumberConstructor' is not
// assignable to parameter of type '(num: number) => string'.
//   Type 'number' is not assignable to type 'string'.(2345)
stringify123(Number);
```

But it accepts the following function call:

```
assert.equal(
  stringify123(String), '123');
```

Result types of function declarations

It’s recommended to annotate all parameters of a function (except for callbacks where more type information is available).

We can also specify the result type:

```
function stringify123(callback: (num: number) => string): string {
  return callback(123);
}
```

TypeScript is good at inferring result types, but specifying them explicitly is occasionally useful.

The special result type `void`

`void` is a special result type for functions: It tells TypeScript that the function always returns `undefined` (explicitly or implicitly):

```
function f1(): void { return undefined } // explicit return
function f2(): void { } // implicit return
function f3(): void { return 'abc' } // error
```

Optional parameters

A question mark after an identifier means that the parameter is optional. For example:

```
function stringify123(callback?: (num: number) => string) {
  if (callback === undefined) {
    callback = String;
  }
  return callback(123);
}
```

TypeScript only lets us make the function call in line A if we make sure that `callback` isn’t `undefined` (which it is if the parameter was omitted).

Parameter default values

TypeScript supports parameter default values:

```
function createPoint(x=0, y=0): [number, number] {
  return [x, y];
}

assert.deepEqual(
  createPoint(),
  [0, 0]);
assert.deepEqual(
  createPoint(1, 2),
  [1, 2]);
```

Default values make parameters optional. We can usually omit type annotations, because TypeScript can infer the types. For example, it can infer that `x` and `y` both have the type `number`.

If we wanted to add type annotations, that would look as follows.

```
function createPoint(x:number = 0, y:number = 0): [number, number] {
  return [x, y];
}
```

Rest parameters

We can also use rest parameters in TypeScript parameter definitions. Their static types must be Arrays:

```
function joinNumbers(...nums: number[]): string {
  return nums.join('-');
}
assert.equal(
  joinNumbers(1, 2, 3),
  '1-2-3');
```

Union types

In JavaScript, variables occasionally have one of several types. To describe those variables, we use *union types*.

For example, in the following plain JavaScript code, `numberOrString` is either of type `number` or of type `string`:

```
function getScore(numberOrString): number {
  if (typeof numberOrString === 'number'
    && numberOrString >= 1 && numberOrString <= 5) {
    return numberOrString
  } else if (typeof numberOrString === 'string'
    && /^\*{1,5}$/.test(numberOrString)) {
      return numberOrString.length;
  } else {
    throw new Error('Illegal value: ' + JSON.stringify(numberOrString));
  }
}

assert.equal(getScore('***'), 3); // OK
assert.throws(() => getScore('')); // not allowed
assert.throws(() => getScore('******')); // not allowed

assert.equal(getScore(3), 3); // OK
assert.throws(() => getScore(0)); // not allowed
assert.throws(() => getScore(6)); // not allowed
```

In TypeScript, `numberOrString` has the type `number|string`. The result of the type expression `S|T` is the set-theoretic union of the types `S` and `T` (while interpreting them as sets).

```
function getScore(numberOrString: number|string): number {
  if (typeof numberOrString === 'string'
    && /^\*{1,5}$/.test(numberOrString)) {
      return numberOrString.length;
  } else if (typeof numberOrString === 'number'
    && numberOrString >= 1 && numberOrString <= 5) {
    return numberOrString
  } else {
    throw new Error('Illegal value: ' + JSON.stringify(numberOrString));
  }
}
```

By default, `undefined` and `null` are not included in types

In many programming languages, `null` is part of all object types. For example, whenever the type of a variable is `String` in Java, we can set it to `null` and Java won't complain.

Conversely, in TypeScript, `undefined` and `null` are handled by separate, disjoint types. We need type unions such as `undefined|string` and `null|string`, if we want to allow them:

```
let maybeNumber: null|number = null;
maybeNumber = 123;
```

Otherwise, we get an error:

```
// @ts-ignore: Type 'null' is not assignable to type 'number'. (2322)
let maybeNumber: number = null;
maybeNumber = 123;
```

Note that TypeScript does not force us to initialize immediately (as long as we don't read from the variable before initializing it):

```
let myNumber; // OK
myNumber = 123;
```

Making omissions explicit

Let's rewrite function `stringify123()`: This time, we don't want the parameter `callback` to be optional. It should always be mentioned. If callers don't want to provide a function, they have to explicitly pass `null`. That is implemented as follows.

```
function stringify123(
  callback: null | ((num: number) => string)) {
  const num = 123;
  if (callback) { // (A)
    return callback(123); // (B)
  }
  return String(num);
}

assert.equal(
  stringify123(null),
  '123');

// @ts-ignore: Expected 1 arguments, but got 0. (2554)
stringify123();
```

Note that, once again, we have to check if `callback` is actually a function (line A), before we can make the function call in line B. Without the check, TypeScript would report an error.

Optional vs. default value vs. `undefined|T`

The following three parameter declarations are quite similar:

- Parameter is optional: `x?: number`
- Parameter has a default value: `x = 456`
- Parameter has a union type: `x: undefined | number`

If the parameter is optional, it can be omitted. In that case, it has the value `undefined`:

```
function f1(x?: number) { return x }

assert.equal(f1(123), 123); // OK
assert.equal(f1(undefined), undefined); // OK
assert.equal(f1(), undefined); // can omit
```

If the parameter has a default value, that value is used when the parameter is either omitted or set to `undefined`:

```
function f2(x = 456) { return x }

assert.equal(f2(123), 123); // OK
assert.equal(f2(undefined), 456); // OK
assert.equal(f2(), 456); // can omit
```

If the parameter has a union type, it can’t be omitted, but we can set it to `undefined`:

```
function f3(x: undefined | number) { return x }

assert.equal(f3(123), 123); // OK
assert.equal(f3(undefined), undefined); // OK

// @ts-ignore: Expected 1 arguments, but got 0. (2554)
f3(); // can’t omit
```

Typing objects

Similarly to Arrays, objects play two roles in JavaScript (that are occasionally mixed and/or more dynamic):

- Records: A fixed number of properties that are known at development time. Each property can have a different type.
- Dictionaries: An arbitrary number of properties whose names are not known at development time. One type per kind of key (mainly: string, symbol).

We’ll ignore objects-as-dictionaries in this blog post. As an aside, Maps are usually a better choice for dictionaries, anyway.

Typing objects-as-records via interfaces

Interfaces describe objects-as-records. For example:

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

We can also separate members via commas:

```
interface Point {
  x: number,
  y: number,
}
```

TypeScript’s structural typing vs. nominal typing

One big advantage of TypeScript’s type system is that it works *structurally*, not *nominally*. That is, interface `Point` matches all objects that have the appropriate structure:

```
interface Point {
  x: number;
  y: number;
}
function pointToString(pt: Point) {
  return `(${pt.x}, ${pt.y})`;
}

assert.equal(
  pointToString({x: 5, y: 7}), // compatible structure
  '(5, 7)');
```

Conversely, in Java's nominal type system, we must explicitly declare with each class which interfaces it implements. Therefore, a class can only implement interfaces that exist at its creation time.

Object literal types

Object literal types are anonymous interfaces:

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

One benefit of object literal types is that they can be used inline:

```
function pointToString(pt: {x: number, y: number}) {
  return `(${pt.x}, ${pt.y})`;
}
```

Optional properties

If a property can be omitted, we put a question mark after its name:

```
interface Person {
  name: string;
  company?: string;
}
```

In the following example, both `john` and `jane` match the interface `Person`:

```
const john: Person = {
  name: 'John',
};
const jane: Person = {
  name: 'Jane',
  company: 'Massive Dynamic',
};
```

Methods

Interfaces can also contain methods:

```
interface Point {
  x: number;
  y: number;
  distance(other: Point): number;
}
```

The type system doesn’t distinguish between methods and properties whose values are functions.

```
interface Num1 {
  value: number;
  square(): number;
}
interface Num2 {
  value: number;
  square: () => number;
}

const num1 = {
  value: 3,
  square() {
    return this.value ** 2;
  }
};
const num2 = {
  value: 4,
  square: () => {
    return num2.value ** 2;
  }
};

const n11: Num1 = num1;
const n21: Num2 = num1;
const n12: Num1 = num2;
const n22: Num2 = num2;
```

However, the distinction is still meaningful for humans: It expresses how we expect properties to be set up and used.

Type variables and generic types

With static typing, we have two levels:

- Values exist at the *base level*.
- Types exist at a *meta level*.

Similarly:

- Normal functions exist at the base level, are factories for values and have parameters representing values. Parameters are declared between parentheses:

  ```
  const valueFactory = (x: number) => x; // definition
  const myValue = valueFactory(123); // use
  ```

- Parameterized types exist at the meta level, are factories for types and have parameters representing types. Parameters are declared between angle brackets:

  ```
  type TypeFactory<X> = X; // definition
  type MyType = TypeFactory<string>; // use
  ```

Example: a container for values

```
// Factory for types
interface ValueContainer<Value> {
  value: Value;
}
// Creating one type
type StringContainer = ValueContainer<string>;
```

`Value` is a *type variable*. One or more type variables can be introduced between angle brackets.

Example: a type-parameterized class

This time, the class `SimpleStack` has the type parameter `T`. (Single uppercase letters such as `T` are often used for type parameters.)

```
class SimpleStack<T> {
  #data: Array<T> = [];
  push(x: T): void {
    this.#data.push(x);
  }
  pop(): T {
    const result = this.#data.pop();
    if (result === undefined) {
        throw new Error();
    }
    return result;
  }
  get length() {
    return this.#data.length;
  }
}
```

When we instantiate the class, we also provide a value for the type parameter:

```
const stringStack = new SimpleStack<string>();
stringStack.push('first');
stringStack.push('second');
assert.equal(stringStack.length, 2);
assert.equal(stringStack.pop(), 'second');
```

Example: Maps

Maps are typed generically in TypeScript. For example:

```
const myMap: Map<boolean,string> = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
```

Thanks to type inference (based on the argument of `new Map()`), we can omit the type parameters:

```
// %inferred-type: Map<boolean, string>
const myMap = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
```

Type variables for functions

Functions (and methods) can introduce type variables, too:

```
function id<T>(x: T): T {
  return x;
}
```

We use this function as follows.

```
id<number>(123);
```

Due to type inference, we can once again omit the type parameter:

```
id(123);
```

A more complicated function example

```
function fillArray<T>(len: number, elem: T) {
  return new Array<T>(len).fill(elem);
}
```

The type variable `T` appears three times in this code:

- `fillArray<T>`: introduce the type variable
- `elem: T`: use the type variable, pick it up from the argument.
- `Array<T>`: pass on `T` to the `Array` constructor.

That means: we don't have to explicitly specify the type `T` of `Array<T>`. It is inferred from parameter `elem`:

```
// %inferred-type: string[]
const arr = fillArray(3, '*');
assert.deepEqual(
  arr, ['*', '*', '*']);
```

Conclusion: understanding the initial example

Let's use what we have learned to understand the piece of code we have seen earlier:

```
interface Array<T> {
  concat(...items: Array<T[] | T>): T[];
  reduce<U>(
    callback: (state: U, element: T, index: number, array: T[]) => U,
    firstState?: U
  ): U;
  // ···
}
```

This is an interface for an Array whose elements are of type `T` that we have to fill in whenever we use this interface:

- method `.concat()` has zero or more parameters (defined via the rest operator). Each of those parameters has the type `T[]|T`. That is, it is either an Array of `T` values or a single `T` value.
- method `.reduce()` introduces its own type variable, `U`. `U` expresses the fact that the following entities all have the same type (which we don't need to specify, it is inferred automatically):
    - Parameter `state` of `callback()` (which is a function)
    - Result of `callback()`
    - Optional parameter `firstState` of `.reduce()`
    - Result of `.reduce()`

  `callback` also gets a parameter `element` whose type has the same type `T` as the Array elements, a parameter `index` that is a number and a parameter `array` with `T` values.

Further reading

- Book (free to read online): "JavaScript for impatient programmers"
- "ECMAScript Language Types" in the ECMAScript specification.
- "TypeScript Handbook": is well-written and explains various other kinds of types and type operators that TypeScript supports.
- The TypeScript repository has type definitions for the complete ECMAScript standard library. Reading them is an easy way of practicing the type notation.