Introduction to Javascript/Typescript

Composing Software

An Exploration of Functional Programming and Object Composition in JavaScript

by Eric Elliott

Javascript vs. Typescript

Typescript is superset of javascript

Provide an optional structural type system for JavaScript.

Typescript and functional programming language

TypeScript is a multi-paradigm programming language and, as a result, it includes many influences from both OOP languages and functional programming paradigms.

It is not a purely functional programming language because, for example, the TypeScript compiler doesn't force our code to be free of side-effects.

TypeScript provides us with an extensive set of features that allow us to take advantage of some of the best features of the world of OOP languages and the world of functional programming languages.

Why we need type:

> Why do I need to pass the type everywhere? That sucks. I can take care about them myself. from Why TypeScript is the best way to write Front-end in 2019, Jack Tomaszewski You may sometimes hear Haskell programming saying: "If it compiles, it must be correct". from Category Theory for Programmers.

Demo: Compare static typed and dynamically typed

Types are optional in typescript

Why we need type in functional programming language?

Types are about composability

Functions are chained from output to input, and the program will not work if the target function is not able to correctly interpret the data produce by the source function

Expression and values

An expression is a chunk of code that evaluates to a value.

```
7;
7 + 1;
'Hello'
```

Variables: var/let/const

var

```
//variables.ts
function f1(): void {
    if (true) {
        var bar: number = 0;
    }
    console.log(bar);
}
function f2(){
    bar = 0
    var bar : number
}
```

Variables: var/let/const

```
function g(): void {
    if (true) {
        let bar: number = 0;
        bar = 1;
    }
    //console.log(bar); // Error
}
```

Variables : var/let/const

Difference between var and let

- Variables declared by var keyword are scoped to the immediate function body (hence the function scope)
- Variables declared by let variables are scoped to the immediate enclosing block denoted by { }
- Function scope and hoisting

Variables : var/let/const

const Variables defined with const behave like let variables, except they cannot be reassigned:

```
function foo(): void {
    if (true) {
        const bar: number = 0;
        //bar = 1; // Error
    }
    //alert(bar); // Error
}
```

Variables : how to select

By default, select the strictest declaration: const.

But the const key word noes not make the variable immutable:

```
const dog = {
  age: 3
}
dog.age = 5
dog = { name: 'biko' }
```

Ternaries expression

if/else or Ternaries

```
var max
if(a>b)
    max = a;
else
    max = b;

const max = a > b ? a : b
```

Ternaries expression

if/else or Ternaries

prefer side-effect free expression, avoid buggy control flow structure.

```
//lisp
(if (eql year 1990) "Hammertime" "Not Hammertime")
```

```
//javascript
ifElse(gte(v,18),"adult","children")
```

Comparisons

- === - Strict Equality Comparison ("strict equality", "identity", "triple equals")
- == - Abstract Equality Comparison ("loose equality", "double equals")

Declare function : named function

```
//function declaration
function greetNamed(name: string): string {
    return 'Hi! ${name}';
}
```

Declare function : function expression

```
const greetUnnamed1 = function(name:string) {
     return 'Hi! ${name}';`
}
```

Declare function : Arrow function

```
const greetUnnamed2 = (name: string) =>{
    return 'Hi! ${name}';
}
```

Declare function : type alias

```
type GreetingFuncType =  (name: string) => string;

let greetUnnamed : GreetingFuncType = {
    return 'Hi! ${name}';
};
```

Declare function : generator

How to iterate over a container: **iterator** is a simple but important design pattern, abstract away the inner structure of a container

```
for(let i of values){
    //processing
}
```

```
for(int i : values){
    //processing
}
```

Question: how to iterator over a range of numbers, for example, from 100 to 1,000,000

generator : Procedure version

```
Iterable<Integer> range(int start, int end){
    List<Integer> ints = new ArrayList<Integer>();
    for (int i = start; i <= end; i++) {
        ints.add(i);
    }
    return ints;
}
```

Problem : how about lazy?

generator : OO version

```
public class IntRangeIterator implements Iterator<Integer> {
  private int nextValue;
  private final int max;
  public IntRangeIterator(int min, int max) {
    this.nextValue = min;
    this.max = max;
  }
  public boolean hasNext() {
    return nextValue <= max;
  }
  public Integer next() {
    if (!hasNext()) {
      throw new NoSuchElementException();
    }
    return Integer.valueOf(nextValue++);
  }
}
```

generator : javascript version

```
function* range(start,end){
    for(let i = start;i<=end;i++)
        yield i;
}
//usage:
for(let i of range(20,30)){
    console.log(i);
}
```

Functions with optional parameters

In typescript, the number of arguments given to a function has to match the number of parameters the function expects.

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

// error, too few parameters
let result1 = buildName("Bob");
// error, too many parameters
let result2 = buildName("Bob", "Adams", "Sr.");
// ah, just right
let result3 = buildName("Bob", "Adams");
```

Functions with optional parameters

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
// works correctly now
let result1 = buildName("Bob");
// error, too many parameters
let result2 = buildName("Bob", "Adams", "Sr.");
// ah, just right
let result3 = buildName("Bob", "Adams");
```

Default parameters

```
function f(firstName: string, lastName?: string) {
    // ...
}
```

and

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
```

Rest Parameters

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none.

```
function buildName(
    firstName: string, ...restOfName: string[]
){
    return firstName + " " + restOfName.join(" ");
}

// employeeName will be "Joseph Samuel Lucas MacKinzie"
let employeeName = buildName(
    "Joseph", "Samuel", "Lucas", "MacKinzie"
);
```

Function overloading

JavaScript is inherently a very dynamic language. It’s not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

Function overloading

```
function test(name:string):string;
function test(age : number):number;

function test(value : string|number) : any{
    switch(typeof value){
        case 'string':
            return `my name is ${value}`;
        case 'number':
            return value + 1;
    }
}
```

Destructuring

- Object destructing
- Array destructing

Destructing : Object destructing

Destructuring simply implies breaking down a complex structure into simpler parts.

```
var rect = { x: 0, y: 10, width: 15, height: 20 };

// Destructuring assignment
var {x, y} = rect;
console.log(x, y); // 0,10
```

Demo: destructing.js : adaptor pattern

Destructing : Array destructing

The left-hand side of the assignment define what values to unpack from the sourced variable

```
function f() {
  return [1, 2];
}

var a, b;
[a, b] = f();
console.log(a); // 1
console.log(b); // 2
```

Destructing : Array destructing

Using array destructing

```
function parseProtocol(url) {
  var parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.*)$/.exec(url);
  if (!parsedURL) {
    return false;
  }
  console.log(parsedURL);
  // [
  //    "https://developer.mozilla.org/en-US/Web/JavaScript",
  //    "https", "developer.mozilla.org", "en-US/Web/JavaScript"
  // ]
  var [, protocol, fullhost, fullpath] = parsedURL;
  return protocol;
}
```

Typescript's Type System

- Types increase your agility when doing refactoring. It's better for the compiler to catch errors than to have things fail at runtime.
- Types are one of the best forms of documentation you can have. The function signature is a theorem and the function body is the proof.

Type are structural

Structural vs. Nominal typing

Checking against the name is nominal typing and checking against the structure is structural typing.

Demo : structuraltyping.ts

Type annotation

For a variable, the type annotation comes after the identifier and is preceded by a colon

1. const identifier = value
2. const identifier : type ;
3. const identifier : type = value;

Type annotation : Primitive type

```
const name: string = 'Steve';
const heightInCentimeters: number = 182.88;
const isActive: boolean = true;
```

Type annotation : Array type

```
const names: string[] = ['James', 'Nick', 'Rebecca', 'Lily'];
```

Type annotation : Function type

function annotation with parameter type annotation and return type annotation

```
let sayHello: (name: string) => string;
// implementation of sayHello function
let sayHello = function (name) {
    return 'Hello ' + name;
};
```

Type annotation : Object type

Also inline annotation

```
let person: {
    name: string;
    heightInCentimeters: number;
};
// Implementation of a person object
person = {
    name: 'Mark',
    heightInCentimeters: 183
};
```

Classes

Typescript can be used to build applications using this object-oriented approach.

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```

Classes

- Inheritance
- Access control: **default public**
- **Readonly modifier**
- Static properties
- Abstract classes
- **Using a class as an interface**

Classes : Readonly modifier

You can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Classes : Using a class as an interface

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

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

Interfaces

An interface can be used as an abstract type that can be implemented by concrete classes, can also be used to define any structure in your TypeScript program

```
let myPoint1: { x: number; y: number; };

interface Point {
    x: number; y: number;
}
let myPoint2: Point;
```

Interfaces

We can define interfaces with same name.

It allows you to mimic the extensibility of JavaScript using interfaces.

```
interface Point {
    z: number;
}
let myPoint2: Point;
myPoint2.z
```

Interfaces

Classes can implement interfaces

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

class MyPoint implements Point {
    x: number; y: number; // Same as Point
}

var foo: Point = new MyPoint();
```

Type Aliases

Create a new name from existing types.

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === "string") {
        return n;
    }
    else {
        return n();
    }
}
```

Union Types

A union type widens the allowable values by specifying that the value can be of more than a single type.

Union Types

```
// Type annotation for a union type
let union: boolean | number; // OK: number
union = 5;
// OK: boolean
union = true;
// Error: Type "string" is not assignable to type 'number | boolean'
union = 'string';
// Type alias for a union type
type StringOrError = string | Error;
// Type alias for union of many types
type SeriesOfTypes = string | number | boolean | Error;
```

Union type

Discriminated Union.

Demo: DiscriminatedUnion.ts

```
interface Square {
    kind: "square";
    size: number;
}

interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
type Shape = Square | Rectangle;
```

Union type

Demo: DiscriminatedUnion.ts

```
function area(s: Shape) {
    if (s.kind === "square") {
        // Now TypeScript *knows* that `s` must be a square ;)
        // So you can use its members safely :)
        return s.size * s.size;
    }
    else {
        // Wasn't a square? So TypeScript will figure out that it must be a Rectan
        // So you can use its members safely :)
        return s.width * s.height;
    }
}
```

Intersection Types

An intersection type combines several different types into a single supertype that includes the members from all participating types

```
interface Skier {
    slide(): void;
}
interface Shooter {
    shoot(): void;
}
type Biathelete = Skier & Shooter;

let b : Biathelete;
b.slide();
b.shoot();
```

Tuple Types

A tuple type uses an array, and specifies the type of elements based on their position

```
let poem: [number , boolean , string];
// OK
poem = [1, true, 'love'];
// Error: 'string' is not assignable to 'number'
poem = ['my', true, 'love'];
```

Tuple used in return values Demo:

Generic

Being able to create a component that can work over a variety of types rather than a single one.

Demo

既保持静态类型检查，又能够避免冗余的代码。

Generic class

```
class GenericAdder<T>{
    constructor(zeroValue: T,add: (x: T, y: T) => number){
        this.zeroValue = zeroValue;
        this.add = add;
    }
    zeroValue: T;
    add: (x: T, y: T) => number;
}
```

Generic class

```
class GenericAdder<T>{
    constructor(zeroValue: T,add: (x: T, y: T) => number){
        this.zeroValue = zeroValue;
        this.add = add;
    }
    zeroValue: T;
    add: (x: T, y: T) => number;
}
```

```
const n = new GenericAdder<number>(0,(x,y)=>x+y);
const s = new GenericAdder<string>("",(x,y)=>x.length+y.length)
```

Generic Constraints

Sometimes we want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have.

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

Generic Constraints

Sometimes we want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have.

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);  // Error: T doesn't have .length
    return arg;
}
```

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    // Now we know it has a .length property,
    // so no more error
    console.log(arg.length);
    return arg;
}
```

Generic Constraints : Type Parameters

在Typescript中，类型中成员(property)的名称可以构成一个类似枚举的类型。

Demo

Generic Constraints : Type Parameters

在Typescript中，类型中成员(property)的名称可以构成一个类似枚举的类型。

Demo

```
const v = {length:10};

const prop = <T>(v:T,key:keyof T) => v[key]
prop(v,"length")
prop(v,"l") //compile error
```

Mapped Types

Mapped types allow you to create new types from existing ones by mapping over property types. Each property of the existing type is transformed according to a rule that you specify. The transformed properties then make up the new type.

Mapped Types

Copy paste and create a new type:

```
interface Options {
    material: string;
    backlight: boolean;
}

// Manually created
interface PickOptions{
    material : string
}
// Manually created readonly interface
interface ReadonlyOptions {
    readonly material: string;
    readonly backlight: boolean;
}
```

Mapped types

```
type PickOptions = {
    [P in "material"] : Options[P];
}
type ReadonlyOptions = {
    readonly [P in keyof Options] : Options[P];
}
```

Generic Mapped types

```
type Pick<T,K extends keyof T> = {
    [P in K] : T[P]
}
type PickOptions = Pick<Options,"material">
```

Generic Mapped types

```
type Pick<T,K extends keyof T> = {
    [P in K] : T[P]
}
type PickOptions = Pick<Options,"material">
```

```
type Readonly<T> = {
    readonly [P in keyof T] : T[P]
}
type ReadonlyOptions = Readonly<Options>
```

Mapped type : exercise

Please define a interface like HitCountOptions using type mapping.

```
interface Options {
    material: string;
    backlight: boolean;
}
//TODO 使用Mapped Type实现下HitCountOptions
interface HitCountOptions {
    readonly material: string;
    readonly backlight: string;
}
const hitCount : HitCountOptions = {
    material : "material",
    backlight : "backlight2"
}
console.log(hitCount.material.length) // 8
hitCount.backlight = "headlight" // compile error
```

Generic : Generic factory

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions.

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```