FLOW VS TYPESCRIPT

TYPE SYSTEMS FOR JAVASCRIPT

Oliver Zeigermann / @DJCordhose

http://djcordhose.github.io/flow-vs-typescript/2016_hhjs.html

PART I: INTRODUCTION

WHY USING TYPE SYSTEMS?

IMHO TYPE SYSTEMS MAKE CODE EASIER TO MAINTAIN

type annotations

- can make code more readable
- can make code easier to analyse
- can allow for reliable refactoring
- can allow for generally better IDE support
- can catch errors early

MOST IMPORTANT SINGLE USE CASE FOR ME

JSON style data structures

- REST payload
- config files
- objects from and to database

as a close second place: *find usages*

TYPESCRIPT

EASE OF USE AND TOOL SUPPORT OVER SOUNDNESS

- http://www.typescriptlang.org/
- By Microsoft (Anders Hejlsberg)
- Based on ES6 (probably ES7/ES8)
- Adds optional type annotations, visibility, and decorators
- Compiler checks and removes annotations
- Latest 1.8 release adds more general sane checks
- External declarations can add type information to pure JavaScript
- Extensive support in WebStorm and Visual Studio Code

FLOW

SOUNDNESS, NO RUNTIME EXCEPTIONS AS GOAL

- http://flowtype.org/
- By Facebook
- *Flow is a static type checker, designed to quickly find errors in JavaScript applications*
- Not a compiler, but checker
- Works without any type annotations
- Very good at inferring types
- If present, type annotations can very easily be removed by babel for runtime

PART II: COMPARISON

AKA WHAT I FIND MOST IMPORTANT

BASICS

TYPESCRIPT

```
let obj: string;
obj = 'yo';
// Error: Type 'number' is not assignable to type 'string'.
obj = 10;
```

```
// types can be inferred (return type)
function sayIt(what: string) {
    return `Saying: ${what}`;
}
const said: string = sayIt(obj);
```

```
class Sayer {
    // mandatory
    what: string;

    constructor(what: string) {
        this.what = what;
    }

    // return type if you want to
    sayIt(): string {
        return `Saying: ${this.what}`;
    }
}
```

FLOW

```
let obj: string;
obj = 'yo';
// Error: number: This type is incompatible with string
obj = 10;
```

```
function sayIt(what: string) {
    return `Saying: ${what}`;
}
const said: string = sayIt(obj);
```

```
class Sayer {
    what: string;

    constructor(what: string) {
        this.what = what;
    }

    sayIt(): string {
        return `Saying: ${this.what}`;
    }
}
```

RIGHT, PRETTY MUCH THE SAME

Those basic features help with documentation, refactoring, and IDE support

NON-NULLABLE TYPES

TALKING ABOUT CORRECTNESS

TYPESCRIPT

```
function foo(num: number) {
    if (num > 10) {
      return 'cool';
    }
}
```

```
// cool
const result: string = foo(100);
console.log(result.toString());
```

```
// still cool?
console.log(foo(1).toString());
```

```
// error at runtime
"Cannot read property 'toString' of undefined"
```

TypeScript does not catch this

FLOW

```
function foo(num: number) {
    if (num > 10) {
        return 'cool';
    }
}

// error: call of method `toString`.
// Method cannot be called on possibly null value
console.log(foo(100).toString());
```

Flow does catch this

But why?

FLOW DOES NOT INFER STRING AS THE RETURN TYPE

The inferred type is something else

```
// error: return undefined. This type is incompatible with string
function foo(num: number): string {
        if (num > 10) {
                return 'cool';
        }
}
```

```
// nullable type: the one inferred
function foo(num: number): ?string {
        if (num > 10) {
                return 'cool';
        }
}
```

```
// to fix this, we need to check the result
const fooed: ?string = foo(100);
if (fooed) {
    fooed.toString();
}
```

NON-NULLABLE TYPES

Types are nullable in TypeScript

Types are non-nullable by default in Flow

Nullable types in Flow also possible

Non-nullable types not yet present in TypeScript
(but there is hope)

GENERICS

AKA PARAMETRIC TYPES

MIGHT BE A BIT SCARY...

CONSIDER THIS TYPE HIERARCHY

```
class Animal {
   name: string;
   constructor(name: string) {
       this.name = name;
   }
}
```

```
class Dog extends Animal {
    // just to make this different from cat
    goodBoyFactor: number;
}
```

```
class Cat extends Animal {
    purrFactor: number;
}
```

GENERIC TYPE INFORMATION

Types can be parameterized by others

Most common with collection types

```
let cats: Array<Cat> = []; // can only contain cats
let animals: Array<Animal> = []; // can only contain animals
```

```
// nope, no cat
cats.push(10);
```

```
// nope, no cat
cats.push(new Animal('Fido'));
```

```
// cool, is a cat
cats.push(new Cat('Purry'));
```

```
// cool, cat is a sub type of animal
animals.push(new Cat('Purry'));
```

UP TO THIS POINT THIS PRETTY MUCH WORKS IN FLOW AND TYPESCRIPT THE SAME WAY ...

… BUT WAIT

TYPESCRIPT

```
let cats: Array<Cat> = []; // can only contain cats
let animals: Array<Animal> = []; // can only contain animals
```

```
// error TS2322: Type 'Animal[]' is not assignable to type 'Cat[]'.
//  Type 'Animal' is not assignable to type 'Cat'.
//    Property 'purrFactor' is missing in type 'Animal'.
// cats = animals;
```

```
// wow, works, but is no longer safe
animals = cats;
```

```
// because those are now all cool
animals.push(new Dog('Brutus'));
animals.push(new Animal('Twinky'));
```

```
// ouch:
cats.forEach(cat => console.log(`Cat: ${cat.name}`));
// Cat: Purry
// Cat: Brutus
// Cat: Twinky
```

FLOW

```
let cats: Array<Cat> = []; // can only contain cats
let animals: Array<Animal> = []; // can only contain animals
```

```
// ERROR
// property `purrFactor` of Cat. Property not found in Animal
// cats = animals;
```

```
// same ERROR
// animals = cats;
```

End of story for Flow

WHY?

- TypeScript
  - parametric types are compatible if the type to assign from has a more special type parameter
  - seems most intuitive
  - allows for obviously wrong code, though
- Flow
  - using generic types you can choose from invariant (exact match), covariant (more special), and contravariant (more common)
  - Array in Flow has an invariant parametric type
  - more expressive
  - harder to understand

PART III: EPILOGUE

SHOULD I USE A TYPE CHECKER?

- there seems to be little or no impact on productivity
- initial effort to introduce a checker is low, though (especially for true for flow)
- but: a type system is a complex thing

My recommendation

- if your project does not live for long: *no*
- if your project is really simple: *no*
- if there is a chance you will need to refactor the thing: *yes*
- if your system is very important or even crucial for the success of your company: *yes*
- if people enter or leave your team frequently: *yes*

WRAP-UP

- TypeScript and Flow have influenced each other heavily
- Basic typings are pretty similar
- Both also support React
- Many more constructs like union, intersection, and array types in both
- Flow can even understand TypeScript declaration files
- TypeScript is a compiler, Flow is a checker
- Flow shoots for soundness, TypeScript for tool support
- Flow has non-nullable types as defaults
- Generics in TypeScript are easier, but less expressive
- Flow's type system is generally more expressive
- Flow written in OCaml, Typescript in Typescript

THANK YOU!

QUESTIONS / DISCUSSION

Oliver Zeigermann / @DJCordhose