Type Systems for JavaScript

Flow and TypeScript

Technical Summit 2016

Oliver Zeigermann / @DJCordhose

Slides: http://bit.ly/ms-typescript

Why using type systems?

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 some (type related) errors early

Anders Hejlsberg@Build2016: *Big JavaScript codebases tend to become "read-only".*

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
- 2.x with major changes released recently

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
- If present, type annotations can very easily be removed by babel for runtime

Basics

Demo

Some basic TypeScript hacking in Visual Studio Code

TypeScript as hacked in VS Code

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

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

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

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

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

Flow looks the same for those examples

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


Nullability

Flow

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

```
console.log(foo(9).toString());
```

```
// error: call of method `toString`.
// Method cannot be called on possibly null value
console.log(foo(9).toString());
```

```
// to fix this, we need to check the result
const fooed: string|void = foo(9);
if (fooed) {
    fooed.toString();
}
```

Types are non-nullable by default in flow

TypeScript

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

```
// same as flow
const fooed: string|void = foo(9);
if (fooed) {
    fooed.toString();
}
```

```
// or tell the compiler we know better (in this case we actually do)
fooed!.toString();
```

Only applies to TypeScript 2.x

Only works when *strictNullChecks* option is checked

All types nullable by default in TypeScript 1.x

`any` type

can be anything, not specified

can selectively disable type checking

```
function func(a: any) {
    return a + 5;
}
```

```
// cool
let r1: string = func(10);

// cool
let r2: boolean = func('wat');
```

- *flow*: explicit any supported, but any never inferred
- *TypeScript*: explicit any supported
- *TypeScript 1.x*: any inferred if no type specified and no direct inference possible
- *TypeScript 2.x*: inference much smarter

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
```

TypeScript allows for birds and dogs to be cats here :)

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

Differences in Generic Types

- 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 master

Union Types

aka Disjoint Unions aka Tagged Unions aka Algebraic data types

to describe data with weird shapes

Depending on some data other data might apply or not

```
type Response = Result | Error; // a disjoint union type with two cases
type Result = { status: 'done', payload: Object };
type Error = { status: 'error', message: string };
```

Implementation both in Flow and TypeScript

```
function callback(response: Response) {
    // given by code completion
    console.log(response.status);
    // does not work,
    // as we do not know if it exists, just yet
    console.log(result.payload); // ERROR
```

```
    switch (response.status) {
        case 'done':
            // this is the special thing:
            // type system now knows, this is a Response
            const result: Result = response;
            console.log(result.payload);
            break;
        case 'error':
            const error: Error = response;
            console.log(error.message);
            break;
    }
}
```

Are Flow and TypeScript like Java/C++/C#?

Not really

- Both
  - optionally typed / any
  - built to match common JavaScript programming patterns
  - type systems more expressive
  - type inference
  - control flow based type analysis
- TypeScript
  - semantically compatible with JavaScript
- Flow
  - just a checker
  - not even a language of its own
  - non-nullability as default

Nominal Typing for Flow classes

```
class Person {
    name: string;
}

class Dog {
    name: string;
}

let dog: Dog = new Dog();

// nope, nominal type compatibility violated
// ERROR: Dog: This type is incompatible with Person
let person: Person = dog;

// same problem
// ERROR: object literal: This type is incompatible with Person
let person: Person = {
    name: "Olli"
};
```

Structural Typing for TypeScript classes

```
class Person {
    name: string;
}

class Dog {
    name: string;
}

let dog: Dog = new Dog();

// yes, correct, as structurally compatible
let person: Person = dog;

// same thing, also correct
let person: Person = {
    name: "Olli"
};
```

Structural vs Nominal Typing

- Nominal Typing: types are compatible when their declared types match
- Structural Typing: types are compatible when their structures match
- Java, C#, C++, C all use nominal typing exclusively
- Flow classes are also treated as nominal types
- TypeScript classes are treated as structural types
- Everything else in both Flow and TypeScript uses structural typing

Structural Typing for both TypeScript and Flow

```
interface NamedObject {
    name: string;
}

// this is fine as nominal typing only applies to Flow classes
let namedObject: NamedObject = dog;

// same thing, also fine
let namedObject: NamedObject = {
    name: "Olli"
};

// not fine in either, missing name
let namedObject: NamedObject = {
    firstName: "Olli"
};
```

Classes in TypeScript

TypeScript has special support for classes

Makes it easier for people coming from Java/C++/C#

- *abstract* classes and methods
- special shortcut constructors (combined definition of fields and initialization)
- interfaces
- public, private, protected
- decorators (aka Attributes in C# and Annotations in Java)
- readonly properties (TypeScript 2) like in C#

Flow does not feature those or any other syntactic sugar, as it is a checker only

Integrations of raw JavaScript files

TypeScript Declaration files

- Core Declarations come with TypeScript compiler
- Needs External Type Declarations for 3rd party libraries
- Managed by Typings tools (*typings install dt~mocha --save*)
- Will be made obsolete by npm (*npm install @types/react --save*)
- If there are no existing declaration files
  - Bad luck
  - Use *allowJs* option to include raw JavaScript (introduced in 1.8)
  - Write a dummy module declaration (vastly simplified in 2.0)

3rd Party Libraries in Flow

- Core Declarations come with Flow Checker, includes React
- Other external declarations are optional
- External Flow Type Definitions
- by far less libraries covered than TypeScript
- some libraries even come with added flow type declarations (e.g. immutable.js)
- Declarations can be added to flow config files
- If there are no existing declaration files: still works, but less powerful

IDE Support

Visual Studio Code

- https://code.visualstudio.com/
- Excellent TypeScript support
- Directly uses Language Service of TypeScript Compiler
- Visual Studio Code itself written in TypeScript

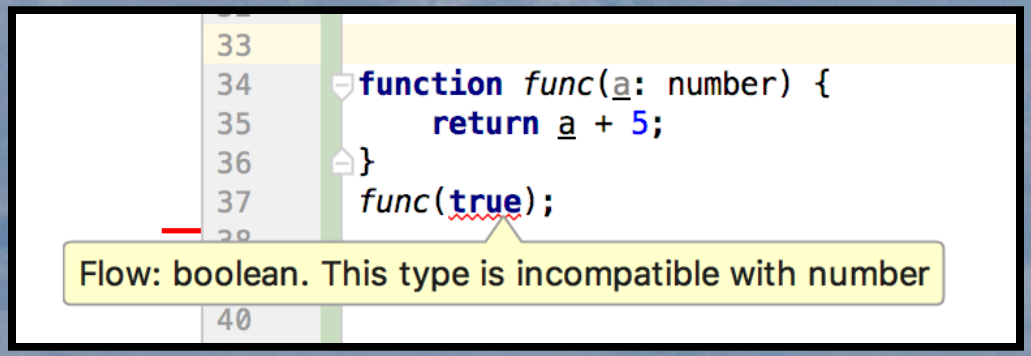
Atom / Nuclide

- https://atom.io/
- https://nuclide.io/ (Atom Package)
- Probably best Flow support

IntelliJ IDEA / Webstorm

Starting from *2016.3*

- Flow: integrated checking and display of messages



- TypeScript: uses Language Service of TypeScript Compiler

Should you use a type checker?

- don't be fooled: checkers do not make your programs error free
- there seems to be little or no impact on productivity
- initial effort to introduce a checker is low, though
- but a type system is a complex thing, it comes at a cost

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*
- you have substantial amount of algorithmic code: *yes*

Where do they excel?

- TypeScript: *supporting people from Java and C# land*
  - more complete IDE support
  - language server
  - large set of 3rd party declaration files
- Flow: *proving types for idiomatic JavaScript*
  - very easy to get started even with existing project
  - more powerful and flexible generics
  - better inference (TypeScript 2.x might catch up)
  - nominal typing for classes

Thank you!

Questions / Discussion

Oliver Zeigermann / @DJCordhose

http://bit.ly/ms-typescript