

Flow: Static Type Checker for JavaScript

Andranik Barseghyan

<https://github.com/Andranik93/Presentation>



Static Type Checker (Flow)

Overview

- Static type checker
- Flow & TypeScript
- What is Flow?
- How does it work?
- Basic Setup
- What does it look like?
Code Example.
- Conclusion
- Resources

Static Type Checker



TypeScript



TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.



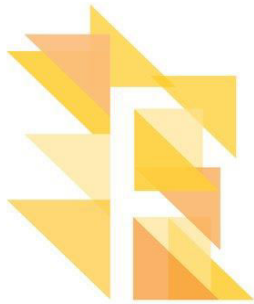
Visual Studio Code



<https://www.typescriptlang.org>



Flow



Flow vs TypeScript



-Checker	-Compiler
-Non-nullable by default	-Nullable by default
-Focused on Soundness	-Focused on Tooling & Scalability
-Written in OCaml: OSX Linux	-Written in TypeScript: any OS
-Works without any annotations	-Great IDE/Editor
-Works out of the box with React	-Used as default by more and more libraries

How dose it work?

Parse Code and generate AST

Inference phase

- . Traverse AST and gather information
- . Create Flow Graph

Evaluation

- . Traverse Flow Graph and find errors

For multiple modules

- . First, create Flow Graph for each module in parallel
- . Then connect Flow Graphs at their touch points

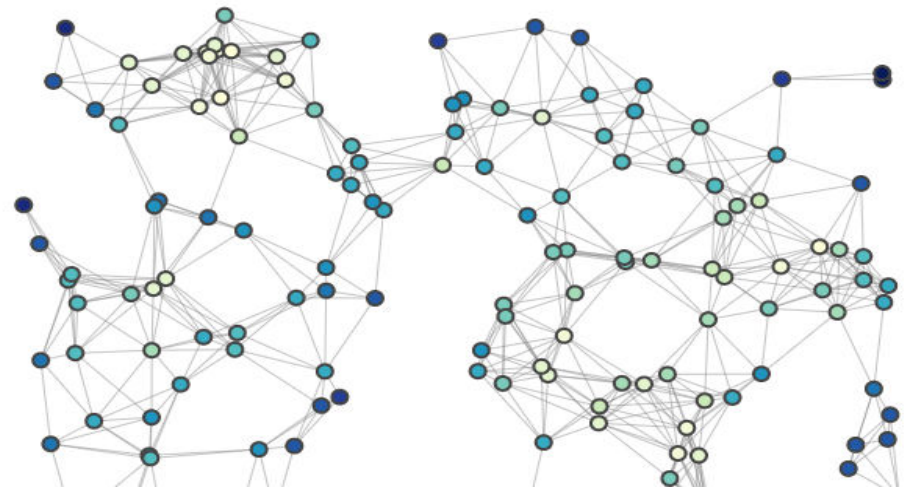
Flow Graph

Not the typical eager approach for type checking

- . Type is decided as late as possible

Possibilities beyond type-checking

- . Tracking data through the application
- . Dead code elimination
(inverse approach of tree-shaking)



Features

Maybe Types	Union and Intersection Types
Array and Tuples	Type Aliases
Classes	TypeOf type
Object	Dinamic Type Tests
Function	Disjoint Unions
Destructuring	Modules
Declaration	



Example

```
1  /**@flow */
2
3  function foo(x) {
4    |   return x * 10;
5  }
6
7  foo('string');
```

```
1  /**@flow */
2
3  function foo(x: number, y) {
4    |   return x * 10 * y;
5  }
6
7  foo(5, 'string');
```

```
1  /**@flow */
2
3  function foo(x: number, y: number): number {
4    |   return x * 10 * y;
5  }
6
7  foo(5, 11);
```



Primitive Types

- Number (Infinity, NaN)
- String
- Boolean
- null
- **undefined** (**void** in Flow types)
- Symbols (not yet supported in Flow)

```
const num: ?number = 11;
```

```
const str: string = 'string';
```

```
const bool: boolean = true;
```



Type

```
1  /**@flow */
2  type bookType = {
3      id: number,
4      author: string,
5      isActive: boolean,
6      other: {
7          title: string,
8          count: number,
9          published: number,
10         date: string | number,
11     }
12 };
13
14 const book: bookType = {
15     id: 1558669888963,
16     author: 'Hermann Hesse',
17     isActive: true,
18     other: {
19         title: 'The Glass Bead Game',
20         count: 450,
21         published: 1942,
22         date: 'Wed, 18 Nov 1942 10:43:12 GMT'
23     }
24 }
```

```
1  /**@flow */
2  /** export.js */
3  export type bookType = {
4      id: number,
5      author: string,
6      isActive: boolean,
7      other: {
8          title: string,
9          count: number,
10         published: number,
11         date: string | number,
12     }
13 };
14
15 /**@flow */
16 /** import.js */
17
18 import { bookType } from './export';
```



Mixed & Any Types

```
1 function stringifyBasicValue(value: string | number) {  
2   return '' + value;  
3 }
```

```
1 function getTypeOf(value: mixed): string {  
2   return typeof value;  
3 }
```

```
1 // @flow  
2 function add(one: any, two: any): number {  
3   return one + two;  
4 }  
5  
6 add(1, 2);    // Works.  
7 add("1", "2"); // Works.  
8 add({}, []);  // Works.
```

The difference is the 'vice-verca': **any** can flow into other types but **mixed** can not.

Any supports covariance and contravariance. It's a super-type and a subtype of all types.

Mixed supports covariance only. It's a super-type and **not** a sub type of all types.



Variables, Array & Tuple Types

```
1 // @flow
2 let foo: number = 1;
3 foo = 2; // Works!
4 // $ExpectError
5 foo = "3"; // Error!
```

```
1 // @flow
2 var fooVar /* : number */ = 1;
3 let fooLet /* : number */ = 1;
4 var barVar: number = 2;
5 let barLet: number = 2;
```

```
1 let tuple1: [number] = [1];
2 let tuple2: [number, boolean] = [1, true];
3 let tuple3: [number, boolean, string] = [1, true, "three"];
```

```
1 let arr1: Array<boolean> = [true, false, true];
2 let arr2: Array<string> = ["A", "B", "C"];
3 let arr3: Array<mixed> = [1, true, "three"]
```

```
1 let arr: number[] = [0, 1, 2, 3];
```

```
1 // @flow
2 let arr1: ?number[] = null; // Works!
3 let arr2: ?number[] = [1, 2]; // Works!
4 let arr3: ?number[] = [null]; // Error!
```



Union Types

```
1 // @flow
2 function toStringPrimitives(value: number | boolean | string) {
3   return String(value);
4 }
5
6 toStringPrimitives(1);      // Works!
7 toStringPrimitives(true);   // Works!
8 toStringPrimitives('three'); // Works!
9
10 // $ExpectError
11 toStringPrimitives({ prop: 'val' }); // Error!
12 // $ExpectError
13 toStringPrimitives([1, 2, 3, 4, 5]); // Error!
```

```
1 // @flow
2 type Success = { success: true, value: boolean };
3 type Failed  = { success: false, error: string };
4
5 type Response = Success | Failed;
6
7 function handleResponse(response: Response) {
8   if (response.success) {
9     var value: boolean = response.value; // Works!
10  } else {
11    var error: string = response.error; // Works!
12  }
13 }
```

```
1 // @flow
2 // $ExpectError
3 function toStringPrimitives(value: number | boolean | string): string {
4   if (typeof value === 'number') {
5     return String(value);
6   } else if (typeof value === 'boolean') {
7     return String(value);
8   }
9 }
```



Intersection Types

```
1 // @flow
2 type A = { a: number };
3 type B = { b: boolean };
4 type C = { c: string };
5
6 function method(value: A & B & C) {
7   // ...
8 }
9
10 // $ExpectError
11 method({ a: 1 }); // Error!
12 // $ExpectError
13 method({ a: 1, b: true }); // Error!
14 method({ a: 1, b: true, c: 'three' }); // Works!
```

```
1 // @flow
2 type A = { a: number };
3 type B = { b: boolean };
4 type C = { c: string };
5
6 function method(value: A & B & C) {
7   var a: A = value;
8   var b: B = value;
9   var c: C = value;
10 }
```

```
1 // @flow
2 type One = { foo: number };
3 type Two = { bar: boolean };
4
5 type Both = One & Two;
6
7 var value: Both = {
8   foo: 1,
9   bar: true
10 };
```



Maybe Types

```
1 // @flow
2 function acceptsMaybeNumber(value: ?number) {
3   // ...
4 }
5
6 acceptsMaybeNumber(42);      // Works!
7 acceptsMaybeNumber();       // Works!
8 acceptsMaybeNumber(undefined); // Works!
9 acceptsMaybeNumber(null);   // Works!
10 acceptsMaybeNumber("42");   // Error!
```

```
1 // @flow
2 function acceptsMaybeNumber(value: ?number) {
3   if (value !== null && value !== undefined) {
4     return value * 2;
5   }
6 }
```



Generic Types

```
1 // @flow
2
3 type IdentityWrapper = {
4   func<T>(T): T
5 }
6
7 function identity(value) {
8   return value;
9 }
10
11 function genericIdentity<T>(value: T): T {
12   return value;
13 }
14
15 // $ExpectError
16 const bad: IdentityWrapper = { func: identity }; // Error!
17 const good: IdentityWrapper = { func: genericIdentity }; // Works!
```

```
1 // @flow
2 function identity<T>(value: T): T {
3   return value;
4 }
5
6 let one: 1 = identity(1);
7 let two: 2 = identity(2);
8 // $ExpectError
9 let three: 3 = identity(42);
```

```
1 // @flow
2 function identity<T>(val: T): T {
3   return val;
4 }
5
6 let foo: 'foo' = 'foo'; // Works!
7 let bar: 'bar' = identity('bar'); // Works!
```



Class (Interface)

```
1 // @flow
2 interface Serializable {
3   serialize(): string;
4 }
5
6 class Foo {
7   serialize() { return '[Foo]'; }
8 }
9
10 class Bar {
11   serialize() { return '[Bar]'; }
12 }
13
14 const foo: Serializable = new Foo(); // Works!
15 const bar: Serializable = new Bar(); // Works!
```

```
1 class MyClass {
2   method(value: string): number { /* ... */ }
3 }
```

```
1 // @flow
2 class MyClass {
3   method() {
4     // $ExpectError
5     this.prop = 42; // Error!
6   }
7 }
```

```
1 // @flow
2 class MyClass {
3   prop: number;
4   method() {
5     this.prop = 42;
6   }
7 }
```



Prop Types

```
1  import React, { Component } from 'react';
2  import PropTypes from 'prop-types';
3
4  class MyComponent extends Component {
5
6      static propTypes = {
7          className: PropTypes.string,
8          children: PropTypes.node,
9          isOpen: PropTypes.bool,
10         /** ..... */
11     };
12
13     static defaultProps = {
14         className: '',
15         children: null,
16         isOpen: false,
17         /** ..... */
18     };
19
20     render() {
21         /** ..... */;
22     }
23 }
24
25 export default MyComponent;
```



Class & Functional Component

```
1 import * as React from 'react';
2
3 type Props = {
4   foo: number,
5   bar?: string,
6 };
7
8 class MyComponent extends React.Component<Props> {
9   render() {
10     this.props.doesNotExist; // Error! You did not define a `doesNotExist`
11
12     return <div>{this.props.bar}</div>;
13   }
14 }
15
16 <MyComponent foo={42} />;
```

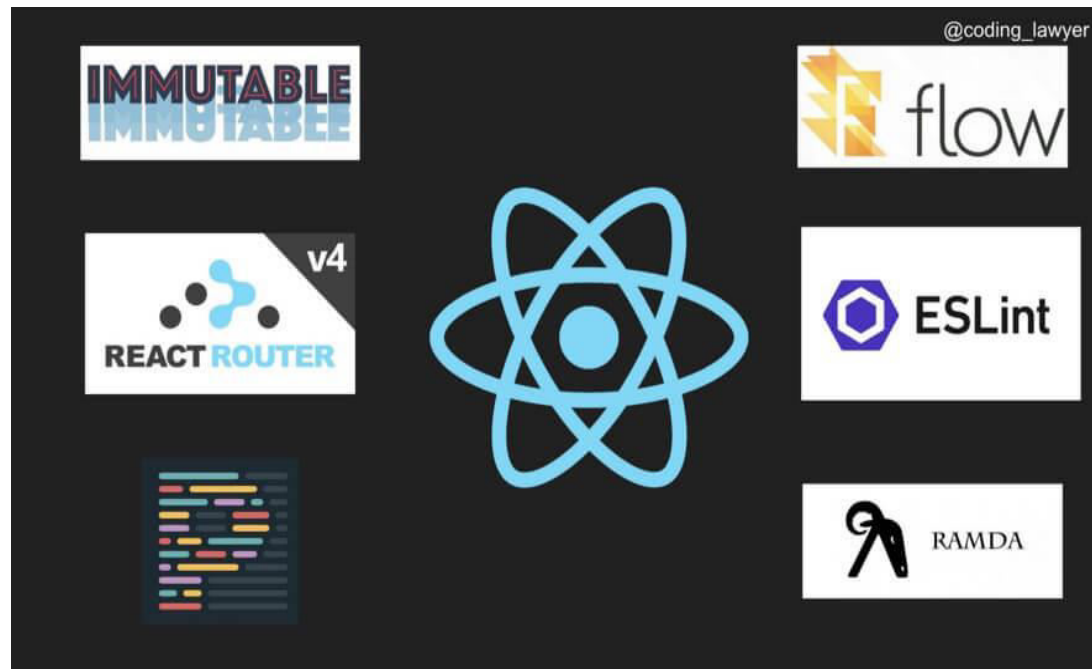
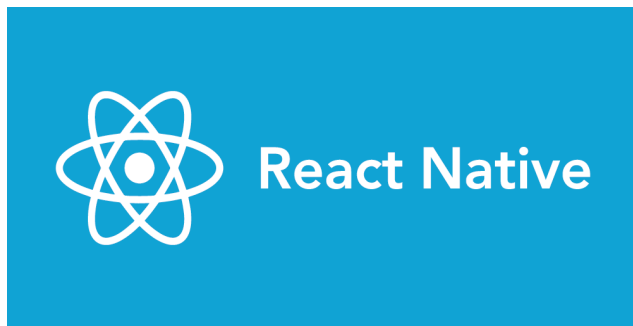
```
1 import * as React from 'react';
2
3 type Props = {
4   foo: number,
5   bar?: string,
6 };
7
8 function MyComponent(props: Props) {
9   props.doesNotExist; // Error! You did not define a `doesNotExist` prop.
10
11   return <div>{props.bar}</div>;
12 }
13
14 <MyComponent foo={42} />
```



Objective Caml Facebook.



OCaml



Conclusion

- Bug & Error
- Documentation
- Refactoring
- Unit Testing



Thank you

