Go with the Flow

Making JavaScript slightly less painful

Owen Harvey

Who here uses JavaScript, either at work or at home?

Who here uses JavaScript, either at work or at home?

Who likes debugging JavaScript errors?

Who here uses JavaScript, either at work or at home?

Who likes debugging JavaScript errors?

Because this is what it feels like for me when I work with JavaScript.

Major problems that I have with JavaScript

- Trying to use something that doesn't exist. Fields, variables, functions, and objects.
- Runtime type errors.
- `null, undefined, false, 0, "0", "", [], [[]], [0], [[0]], ...`
  Strict equality checks only get you so far.
- Refactoring

What to do?

What to do?

Linters help. They will stop you from doing obviously dumb things.

What to do?

Linters help. They will stop you from doing obviously dumb things.

What they don't do, is tell you when you are trying to do something that is syntactically fine, but logically insane.

Can you see the problem in this code?

```
let a = {
  foo: 1
};

let a = _.map(
  a,
  function(e) {
    return e + 1;
  }
);

console.log(a.foo + 1);
```

Can you see the problem in this code?

```
let a = {
  foo: 1
};

let a = _.map(
  a,
  function(e) {
    return e + 1;
  }
);

console.log(a.foo + 1);
```

Neither can I.

This is why we need a type system.

JavaScript type systems, and transpilers.

- Flow: Facebook
- TypeScript: Microsoft
- Purescript, GHCJS, et al. if you want to get fancy.
- Many others, things move fast and people like to use languages they know.

Flow marketing

- Type inference.
- Nullable types.
- Sum types.
- Structurally typed functions and objects.
- Nominally typed classes.
- Optional fields.
- Tuples.
- Polymorphism: Parametric, Ad-hoc, and Bounded.
- Invariant, covariant, and contravariant types and variables.
- Inline and comment annotations.
- Built in utility types.

Flow marketing

- Type inference.
- Nullable types.
- Sum types.
- Structurally typed functions and objects.
- Nominally typed classes.
- Optional fields.
- Tuples.
- Polymorphism: Parametric, Ad-hoc, and Bounded.
- Invariant, covariant, and contravariant types and variables.
- Inline and comment annotations.
- Built in utility types.

- No higher kinded types.
- No pattern matching.
- No exhaustive sum types.
- Unsafe collection access.

Syntax 101

There are two ways that you can annotate your source with Flow.

Inline types, and comment types.

Through this talk I'll be using inline types as I find them more pleasant to work with.

Syntax 101

There are two ways that you can annotate your source with Flow.

Inline types, and comment types.

Through this talk I'll be using inline types as I find them more pleasant to work with.

All Flow typed files start with a comment to tell the type checker that this is a file it can work over. Either of the following comments can be placed at the top of a file, before any code.

```
// @flow
/* @flow */
```

Inline types

Inline types must be stripped before you run your code.

Inline types

Inline types must be stripped before you run your code.

```
// Object type declaration.
type A = { a:string };
// Type alias
type B = number;
// Function declaration, with an explicit return type.
function f(a:A): void { ... }
// Explicitly typed variable assignment.
let a: A = { a: "asd" };
// Implicitly typed variable assignment.
let b = f(a);
```

Inline types

Inline types must be stripped before you run your code.

```
// Object type declaration.
type A = { a:string };
// Type alias
type B = number;
// Function declaration, with an inferred return type.
function f(a:A): void { ... }
// Explicitly typed variable assignment.
let a: A = { a: "asd" };
// Implicitly typed variable assignment.
let b = f(a);
```

Stripped output

```
// Object type declaration.

// Type alias

// Function declaration, with an inferred return type.
function f(a  )       { ... }
// Explicitly typed variable assignment.
let a    = { a: "asd" };
// Implicitly typed variable assignment.
let b = f(a);
```

Comment types

Good if you have an existing build system that is annoying to extend.

Inline comments that start with `:` are treated as type annotations.

```
function f(a/*: string*/)/*: void */ {};
```

A block comment that starts with `::` will be treated as if it were inline type annotations.

```
/*::
type Foo = { foo: string }
*/

class Foo {
  /*:: bar: string; */
}
```

Basic types

Type	Example
`string`	`'Hello, World'`, `"Foo Bar Baz"`
`number`	`1`, `1.0`, `1e10`, `0b10101`, `0765`, `0xAB3`
`boolean`	`true`, `false`
`null`	`null`
`void`	`undefined`

Basic types

Type	Example
`string`	`'Hello, World'`, `"Foo Bar Baz"`
`number`	`1`, `1.0`, `1e10`, `0b10101`, `0765`, `0xAB3`
`boolean`	`true`, `false`
`null`	`null`
`void`	`undefined`
`Function`	`function f() {}`, `() => {}`, `function() {}`
`Object`	`{}`, `{a : null}`
`Array`	`[]`, `[1]`
`mixed`	`""`, `null`, `1`, `function () {}`
`any`	Same as mixed, but also is a subtype of everything.

Optional types

Optional types are denoted with a leading `?`.

```
?number
?MyType
```

Optional fields use a trailing `?`.

```
{ a? :  string } // if a exists, it will be a string
{ a  : ?string } // a exists, but might be null or undefined.
{ a? : ?string } // a may exist, and if so, it might be null or undefined.
```

Type Casting

Values and expressions can be cast to a different type.

This is occasionally required if you are using libraries that don't have type declarations, or you are trying to do something that you know is safe, but are having trouble convincing Flow that everything is fine.

```
type Foo = ...;

type Bar = ...;

function mkBar():Bar { ... };

let x : Foo = ( mkBar(): Foo );
```

Functions

Function declaration follows on from the convention that the type of a thing comes after the thing.

```
let f = function(a: string, b: ?number, c?:Object): ?mixed { ... };

function g(a: Foo, b: Bar, ...rest: string[]): void { ... };
```

Functions

Function declaration follows on from the convention that the type of a thing comes after the thing.

```
let f = function(a: string, b: ?number, c?:Object): ?mixed { ... };

function g(a: Foo, b: Bar, ...rest: string[]): void { ... };
```

If you leave out the type of a parameter, or the return type of the function, Flow will try to infer the type. However, it is best practice to always annotate.

```
function f() { return "string"; };

f() + {};
/*
9: f() + {}; // Compile error
         ^ object literal. This type cannot be added to
9: f() + {}; // Compile error
   ^ string
*/
```

Functions

Arrow functions have a slightly different syntax

```
let f = (a:number, b:string, ...c:Object[]): Object => { ... };
```

And to specify just the type of a function the following is used

```
(a:number, b:string, ...c:Object[]) => Object; // declaration with names
(number, string, ...:Object[]) => Object;      // declaration with just types.
```

The type of the `this` variable for a function is automatically inferred, and shoud not be annotated.

Objects

Objects in Flow have structural subtyping.

In other words, if it has the correct fields, you can use a different type.

Objects

Objects in Flow have structural subtyping.

In other words, if it has the correct fields, you can use a different type.

```
type A = {
  a: string,
  b: number
};

type B = {
  a: string,
  b: number,
  c: Object
};

let b = {
  a: "a",
  b: 1,
  c: {}
};

function f(a:A): void { ... };

f(b); // Type checks
```

Classes

Classes are nominally typed. Even if the implementation is identical, they are different. Think of haskell's `newtype`.

Classes

Classes are nominally typed. Even if the implementation is identical, they are different. Think of haskell's `newtype`.

```
class A {
  a: string
};

class B {
  a: string
};

let b = new B();

function f(a:A) { ... };

f(b); // Error
```

Classes

Class members must be declared before they can be used, either with a type, or a value.

Type annotation is similar to object types, except you use `;` as the field separator.

```
class Foo {
  foo: string;
  bar = 1;
  func() {
    return [this.foo, this.bar];
  };
}
```

Classes

Class members must be declared before they can be used, either with a type, or a value.

Type annotation is similar to object types, except you use `;` as the field separator.

```
class Foo {
  foo: string;
  bar = 1;
  func() {
    return [this.foo, this.bar];
  };
}
```

Additionally, classes are able to extend each other, prototypically inheriting methods and properties.

```
class A { ... };
class B extends A { ... };
```

Arrays

Similar notation to Java/C#.

Append `[]` on the end of your type, or use `Array<T>`

```
let a: Array<number> = [0, 1, 2];

let b: string[] = [
  "Foo",
  "Bar",
  "Baz"
];
```

Arrays

Similar notation to Java/C#.

Append `[]` on the end of your type, or use `Array<T>`

```
let a: Array<number> = [0, 1, 2];

let b: string[] = [
  "Foo",
  "Bar",
  "Baz"
];
```

Array access is unsafe, and you should always check the value returned.

Flow doesn't mark these values as optional, you have to remember.

Tuples

Tuples are represented as fixed length, heterogeneous lists.

```
let t: [number, string] = [1, "a"];

type Foo<A, B> = [A, B, number];
```

Tuples

Tuples are represented as fixed length, heterogeneous lists.

```
let t: [number, string] = [1, "a"];

type Foo<A, B> = [A, B, number];
```

For tuple types to match, both type declarations must have the same length.

Tuples do not match arrays, as the array length is unknown.

```
type A = [number];
type B = [number, number];
type C = [number, number, string];
type D = number[];

let t: B;

t = [1];         // Error
t = [1,2];
t = [1,2,"a"];   // Error
t = ([1,2] : D); // Error
```

Tuples

Array methods that mutate lists are not valid operations on tuples.

```
let t: [string, string] = ["a", "b"];
t.splice(1,0); // Type error.
```

Value access at a specific index, will return a variable of the type at that index.

```
let t: [string, number] = ['a', 1];
t[0] // string
t[1] // number
```

Computed access of the tuple will return return a value with a sum type of all types stored in the tuple.

```
let t: [number, boolean, string] = [1, true, "three"];

function getItem(n: number) {
  let val: number | boolean | string = tuple[n];
};
```

Maps

Maps are objects where the field names aren't known, but the type of the index is.

```
type A = {[string]: number };
type B = {[number]: number };

let a : A = {
  "a": 1,
  "b": 2
};
```

Map access is also unsafe.

Parametric Polymorphism

Again, similar to Java/C# syntax.

```
type Foo<T> = { foo : T }

type Bar<T> = { bar: Foo<T> };

type Baz<S, T> = {
  baz1?: S,
  baz2: ?T
};
```

Bounded Polymorphism

Because Flow is structurally subtyped, you can write code that assumes a variable will exist, and pass in anything that has a matching element.

```
function doSomething(a: {x:number}): {x:number} { ... };

doSomething({x:1, y:'a'});
doSomething({a:1, b:2, c:3, ... x:24, y:25, z:26});
```

But these declarations aren't able to make polymorphic claims about their output.

Bounded Polymorphism

To solve this issue, Flow allows you to give types to generics, preserving all of the type information throughout the function body, and in the return type.

```
type X = {x:number};
type Y = number;
type Foo = X | Y;

function foo<A: Foo>(a:A, b:A):A { return a; };

var a: Y = foo({x:1}, 1);
           ^ object literal. This type is incompatible with
       ^ number

var b: X = foo({x:1}, 1);
           ^ number. This type is incompatible with
       ^ object type
```

Ad hoc Polymorphism

In the running theme of C# and Java similarities, flow supports structural interfaces for classes.

```
interface HasA { a: number; }
class A {
  a: number;
  getA() { return this.a; };
}
class B {
  a: number;
  getA() { return this.a; };
}

var a: HasA = new A();
var b: HasA = new B();
a = b;

var c: A = new B();
           ^ B. This type is incompatible with
       ^ A
```

You can also use `implements` to explicitly list the interfaces a class has.

```
class C implements A, B { ... }
```

Variance

Class Inheritance.

Sub-classes that override super-class methods need to either have the same type signatures, or meet the following conditions.

- Method returns types must be equally, or more specific.
- Method parameters must be equally, or less specific.

```
class A {
  f(a: {x: number}): Object
}
class B extends A {
  f(a: Object): {y: string} { ... };           // Ok
  f(a: {x: number, y: string}): mixed { ... }; // Error
}
```

Variance

Object Property Variance

Object properties are invariant, as they can be both written to, and read from. However, not all properties will be written to, and not all fields will be read from.

Functions that only read from their parameters are able to be covariant (accepts subtypes) in those parameters.

Properties that are only written to can be marked as contravariant (accepts supertypes).

Covariance is denoted with a prefixed `+`, while contravariance uses `-`.

Variance

Object Property Variance

```
// f() is covariant over o
function f(o: {+p: ?string}): number {
  return o.p ? o.p.length : 0;
}

var o: {p: string} = {p: ""};
f(o); // no type error!

// g() is contravariant over o
function g(o: {-p: string}): void {
  o.p = "default";
}
var o: {p: ?string} = {p: null};
g(o);
```

Open and Closed Object types

Open types can have other fields defined when they are passed to a function, you just can't touch them and keep Flow happy without checking for them first.

```
type Foo = { a: number };
let a: Foo = { a: 1, b: "B" };

function foo(f: Foo): string {
  return f.b; // Compile error! b isn't defined in Foo.
}

function bar(f: Foo): string {
  if(f.b != null) { ... } // b can be used in this if block
  ...
}
```

Open and Closed Object types

Closed types can only have the fields described in their type.

```
type Foo = {| a: number |};
let a: Foo = {
  a: 1,
  b: "B"
}; // Type error
```

Sum Types

Sum, or Union, types come in four flavours.

The simplest are a union of literal values.

```
type Sum1 = "A" | "B";
```

The next are a sum of other types.
When using a value you must test for the type.

```
type Sum2 = ?string | number | Object;
```

Sum Types

Disjoint, or tagged, unions require a common field that can be used to determine what type you are working with.
Structural subtyping isn't good enough because open types can have arbitrary extra fields.

```
type A = {
  tag:"A",
  a:any
};
type B = {
  tag:"B",
  b:any
};
type C = {
  tag:"C",
  c:any
};
type D = A | B | C;
```

Sum Types

Disjoint unions with exact types do not require a common field,

```
type W = {|
  w: string
|};
type X = {|
  x: Object
|};
type Y = {|
  y: number
|};
type Z = W | X | Y;

function(z:Z): string {
  if(z.w != null) {
    return z.w; // At this point Flow knows that 'w' exists, so z must be a W
  } else if(z.x != null) {
    return JSON.stringify(z.x);
  } else if(z.y != null) {
    return "number";
  } else {
    return "";
  }
}
```

Utility Types

Flow comes with a set of built in utility types.

The docs suggest that these types are useful when working with React.

Type	Use
`$Keys<T>`	Sum type of key names from a type.
`$Diff<A, B>`	Set difference of object types `A` and `B`.
`Class<T>`	Given a type T representing instances of a class C, the type Class is the type of the class C
`$PropertyType<T, X>`	Type of property `x` in `T`.
`*`	Type inference placeholder.
`$Exact<T>`	Synonym for exact object type.
`$ObjMap<T, F>`	The type obtained by taking the type of the values of an object and mapping them with a type function.

There are a couple more, but they are listed as being under development.

Modules and Imports

Types can be easily shared between files by using Flow's module system, which as a similar syntax to ES6 modules.

```
// exports.js.flow
export type Foo = string;
export interface Bar = { ... };
export class Baz {};
export const aString = "a string";

// import.js
import type Foo          from "./exports";
import type { Foo, Bar } from "./exports";
import typeof aString    from "./exports";
import typeof { Baz }    from "./exports";
```

Import paths can either be relative or absolute.

Flow's config supports a macro syntax that allows you to replace a pattern with the project root.

```
[options]
module.name_mapper='^/' -> '<PROJECT_ROOT>/src/'
```

Library Definitions

By default Flow looks a directory called `flow-typed` in the project root for definitions that should be available globally in the project.

This allows global variables and functions to be given types once, without importing definitions in all files.

The project Flow-Typed acts as the central repository for library definitions, and tools to install them.

Library Definitions

Global Declarations

When you want something to be declared globally, the following syntax is used.

```
// flow-typed/**/*.js
declare type T = { z: Function };
declare type S = T;
declare function f(a:Object):mixed;
declare class Class {
  a: number;
  f: (number, string, ?Object) => boolean;
};
declare var   x: string;
declare const y: object;
```

Library Definitions

Modules

Library definitions can also declare modules, to avoid polluting the global namespace. Both CommonJS and ES style modules are supported, however it is an error to mix the syntax in the same module.

Definitions without an `export` will be available throughout the module body, but will not be exposed externally.

Modules use the same import syntax as `.js.flow` files.

```
import type { Foo } from "Bar";
```

Library Definitions

Modules

```
declare module "es-module" {
  // Non-exposed
  declare class Z { ... };
  // Named exports
  declare export type A = ...;
  declare export function f(a:string, b:number): void;
  // Default export
  declare export default A;
}


declare module "common-js-module" {
  declare class Z { ... };
  declare module.exports: {
    A: { ... };
    f(a:string, b:number): void;
  }
}
```

Checking and Stripping Types

You have a few options based on what you are doing.

Build System	Tool
Babel	babel-preset-flow
Gulp	gulp-flowtype
Grunt	grunt-flow-type-check
Webpack	flow-webpack-plugin
Browserify	flowify
CLI	flow-bin
CLI	flow-remove-types

Not so Great Areas

- Sometimes flow gets confused about empty exact objects

```
type Foo = {||} | {| foo:number |};
let a: Foo = {}; // unknown object literal
```

- There is no exhaustive pattern matching when checking sum types.
- Module declarations and imports can be tricky to get right. If you don't Flow can infer types incorrectly, or default to `any`.
- No higher kinded types. There is a hacky method of approximating them.

Not so Great Areas

- `implements` is not supported in module class declarations.
- The number of available definition files is rather low.
  FlowTyped has ~277 packages
  DefinitelyTyped has ~3225 packages.

  There is a project to automatically convert Typescript definitions to flow definitions, but it is still under development.
  https://github.com/joarwilk/flowgen
- The official documentation can be lacking at times.
  The Flow blog often contains information on new features.
  https://flow.org/blog/
  #flowtype on Freenode