







flow

# Outline

-  Introduction to Flow
-  Flow Types
-  Flow Server
-  Project Setup and Example
-  React Support
-  Library Definitions



# Static vs. Dynamic

- **Types** specify the conditions under which code such as a function or class will run
- In **static** programming languages, type errors are detected at compile-time
- In **dynamic** programming languages, type errors are typically detected at runtime
- **Type checking tools** for dynamic languages allow detection of type errors before runtime
  - such as **Flow** and **TypeScript**





# Why Use Types?

- Can find type errors before runtime
  - more convenient than waiting until runtime
- Types document expectations about code
  - types of variables, object properties, function parameters, and function return types
  - comments can be used instead, but those
    - are more verbose
    - tend to be applied inconsistently
    - easily go out of date when code is updated
- Increases refactoring confidence
  - don't have to wonder what assumptions callers made about supported types
- Removes need to write ...
  - error checking code for type violations
  - type-related unit tests
- Editor/IDE plugins can use types to highlight issues and provide code completion



# Why Avoid Types?

- Takes time to ...
  - learn type syntax
  - master applying them
- Makes code more verbose
  - but also less verbose since there is no need to write error checking code for type violations
- Can hamper prototyping and rapid development
  - developers can lose focus when distracted by having to satisfy a compiler or type checker

# When to Use Types

- Use types when
  - application is large, complex, or critical
  - expected lifetime of code is long and refactoring is likely
  - code will be written and maintained by a team of developers
- Avoid types when
  - the conditions above are not present



# Flow Overview

- “A static type checker, designed to find type errors in JavaScript programs”
- Open source tool from Facebook
  - implemented in OCaml
  - <http://flow.org>
- Catches many errors without types
  - using **type inference** and **flow analysis**
  - “precisely tracks the types of variables as they flow through the program”
- Catches attempts to reference properties on undefined or null references
- Can gradually add types
  - but exported values must have type annotations
    - ex. exported functions must have defined parameter and return types
- Most ES6+ features are supported
  - for a list, see <https://github.com/facebook/flow/issues/560>
- Supports React and JSX

For runtime type checking using Flow syntax, consider babel-plugin-tcomb  
<https://github.com/gcanti/babel-plugin-tcomb>

already supports async/await



# TypeScript Overview

- Competing tool from Microsoft
  - implemented in TypeScript
- Superset of JavaScript syntax adding more than just types
  - class properties can be `readonly`
  - class members can have `public`, `private`, and `protected` modifiers
  - constructor parameter properties, decorators, enums, interfaces, mixins, namespaces (global objects), and more
- Compiles to JavaScript
- Use file extension `.ts` instead of `.js`
- Performs type checking and transpiling (from one version of JS to another)
  - Flow only focuses on type checking
  - with Flow, transpiling is typically handled by Babel
  - good because new JS features generally land in Babel (through plugins) before TypeScript

Paul Graham's "blub paradox"  
<http://www.paulgraham.com/avg.html>



# Comparing Flow & TypeScript

- Type definitions for libraries
  - TypeScript has a repository of type definitions for JavaScript libraries called “**DefinitelyTyped**”
  - Flow has a similar repository called “**flow-typed**”, but it has far fewer entries
  - these allow your code to use third party JavaScript libraries that do not have type annotations and have type checking of those uses
  - as of May 21, 2017, Definitely Typed had types for over **2092** packages, while flow-typed had **570** and many of those are just for different versions of the same package
  - can **generate Flow type declaration files** from TypeScript `.d.ts` files
    - see <https://github.com/joarwilk/flowgen>

```
npm install -g flowgen
flow-gen name.d.ts -o name.flow.js
```
- Both have editor/IDE integrations
- Flow has goal of remaining compatible with TypeScript syntax



# Installing Flow

- Options for installing Flow in a project are described at <https://flow.org/en/docs/install/>
  - can covered later here
- To install globally,  
`npm install -g flow-bin` or use `yarn`
- To get version installed  
`flow version`
- To get help  
`flow --help`



# Running Flow

- To enable type checking a file

- add one of these comments at top

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

**It's easy to forget  
to do this!**

- files without this are not type checked

- To run on a single file instead of all files in project

- `flow check-contents < file-path`
- doesn't require comment at top

I created a bash script  
alternative that can be run  
with "`flow1 file-path`".


- To run on multiple files in a project

- described later



- To get suggested type annotations for a given file
  - `flow suggest file-path`
  - if everything is good, only the file path is output (weird)
  - highly recommended to use this to verify annotations
- Example

```
function rectangleArea(width, height) {  
  return width * height;  
}  
  
const area = rectangleArea(100, 50);  
console.log('area =', area);
```

```
 flow suggest untyped.js  
--- old  
+++ new  
@@ -1,6 +1,6 @@  
-function rectangleArea(width, height) {  
+function rectangleArea(width: number, height: number) : number {  
  return width * height;  
}  
  
-const area = rectangleArea(100, 50);  
+const area: number = rectangleArea(100, 50);  
  console.log('area =', area);
```

doesn't really output  
code in color



# Intro Executing Code With Types ...

- Node.js cannot directly run code that is annotated with Flow types
  - likewise for web browsers (covered later)
- Simple approach for individual files
  - `npm install -g flow-remove-types`
  - installs both `flow-node` and `flow-remove-types` executables
  - to remove types and execute code
    - `flow-node file-path` for Typescript, see <https://github.com/TypeStrong/ts-node>
  - to generate a file without types
    - `flow-remove-types --pretty file-path > new-file-path`
    - `node new-file-path`
  - either way, input files must contain `// @flow`

replaces types with spaces;  
`--pretty` removes whitespace



- Comment approach
  - tedious and ugly, but eliminates need to remove types
  - surround types with `/* : */`
  - surround type aliases with `/* :: */`
- Project-based approach
  - alternative to processing individual files
  - described later



# Flow Analysis Example

flow-analysis-wo-types.js

```
// @flow  
  
function product(n1, n2) {  
  return n1 * n2;  
}  
  
console.log(product(2, 'foo'));
```

```
4:    return n1 * n2;  
      ^^ string. The operand of an arithmetic operation must be a number.
```

TypeScript does not catch this!



# Uninitialized References

uninitialized.js

```
// @flow

function getLastInitial(person) {
  const {lastName} = person;
  return lastName ? lastName[0] : '';
}

const person = {
  firstName: 'Richard',
  middleName: 'Mark',
  lastName: 'Volkman'
};

console.log(getLastInitial(person)); // good; outputs "V"

let p;
console.log(getLastInitial(p)); // error
```

```
16: console.log(getLastInitial(p)); // error
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ function call
4:   const {lastName} = person;
      ^^^^^^^ property `lastName`. Property cannot be accessed on possibly undefined value
4:   const {lastName} = person;
      ^^^^^^ uninitialized variable
```

# Flow Types

- Details are at <http://flowtype.org/docs/quick-reference.html>
- The following slides provide an overview
- Types are specified by appending a colon and type description
  - can specify on variables, properties, parameters, and functions (for return type)
  - ex. `let score: number = 0;`
  - type can be inferred here, but it's only `number` if no other type is ever assigned to this variable



# Maybe Type

- By default, **null** and **undefined** are not allowed by any type
  - to allow, precede type with ?
  - ex. `let score: ?number;`
  - Flow calls this a “**maybe type**”
- TypeScript
  - TypeScript 2 has a setting called “strictNullChecks” that causes it to behave in the same way, but this is not the default behavior

# Types

# Builtin Types

defined in files in <https://github.com/facebook/flow/blob/master/lib/>

- Flow understands ...
- All **built-in** JavaScript constants, functions, objects, and classes defined in `core.js`
  - EX. `Array`, `console`, `Date`, `*Error`, `Function`, `JSON`, `Map`, `Math`, `Promise`, `RegExp`, `Set`, `String`, ...
- Types from the browser Document Object Model (**DOM**) defined in `dom.js`
  - EX. `Document`, `Element`, `*Event`, `HTML*Element`, `Image`, `Node`, `Text`, ...
- Types from the **browser API** defined in `bom.js`
  - EX. `GeoLocation`, `History`, `Location`, `Navigator`, `Request`, `Response`, `Screen`, `SharedWorker`, `WebSocket`, `Worker`, `XMLHttpRequest`, ...
- Types from the **Node.js** standard library defined in `node.js`
  - EX. `Buffer`, `events`, `fs`, `http`, `https`, `net`, `os`, `path`, `process`, `querystring`, `stream`, `url`, ...
- Types from **React** defined in `react.js`
  - EX. `Synthetic*Event` classes
- and more

Don't need to add type annotations for these.  
For example,  
`const fs = require('fs');`  
is enough.



# Basic Types

- primitives: `boolean`, `number`, `string` number includes `Infinity` and `NaN`
- wrappers: `Boolean`, `Number`, `String` - rarely used
- `null`: only matches JavaScript `null` value
- `void`: type of `undefined` and functions that don't return anything
- Specific values (literals) - rarely used
  - ex. `true`, `7`, `'foo'`
- `any`: means any type is allowed
  - typically used when being too lazy to specify the proper type
- `mixed`: similar to `any`, but must perform runtime type checks before using value
  - preferred over `any`

a.k.a type refinements

```
// @flow
function foo(v: mixed) {
  if (typeof v === 'number') return v * 2;
  if (typeof v === 'string') return v.length;
  return v;
}
```

- In JavaScript without types, passing too few arguments results in parameters being set to `undefined`
- With Flow, an error is produced when these parameters have types that do not allow `undefined`
- Still okay to pass more arguments than are expected 

an error in TypeScript

```
// @flow
function product(n1: number, n2: number) {
  return n1 * n2;
}

console.log(product(2)); // error
```

```
7: console.log(product(2));
               ^^^^^^^^^ function call
7: console.log(product(2));
               ^^^^^^^^^ undefined (too few arguments, expected default/rest parameters).
This type is incompatible with
3: function product(n1: number, n2: number) {
               ^^^^^ number
```



# Basic Types Example

basic-types.js

```
// @flow

function getChars(text: string, count: number, fromStart: boolean) {
  return fromStart ? text.substring(0, count) : text.substr(-count);
}

console.log(getChars('abcdefg', 3, true)); // good; outputs 'abc'
console.log(getChars('abcdefg', 3, false)); // good; outputs 'efg'
console.log(getChars(3, false, 'foobar')); // error
```

```
9: console.log(getChars(3, false, 'foobar')); // error
                        ^ number. This type is incompatible with the expected param type of
3: function getChars(text: string, count: number, fromStart: boolean) {
    ^^^^^^ string

9: console.log(getChars(3, false, 'foobar')); // error
                        ^^^^^ boolean. This type is incompatible with the expected param type of
3: function getChars(text: string, count: number, fromStart: boolean) {
    ^^^^^^ number

9: console.log(getChars(3, false, 'foobar')); // error
                        ^^^^^^^ string. This type is incompatible with the expected param type of
3: function getChars(text: string, count: number, fromStart: boolean) {
    ^^^^^^ boolean
```

Found 3 errors

# Type Aliases

- Useful for custom types that are used multiple times
  - most are
- Often used for
  - objects with certain properties that aren't instances of a specific class
    - can describe required properties, but extra properties are allowed known as "width subtyping"
  - callback function signatures
- To define a type alias
  - `type SomeNameType = some-type;`
  - common convention is for type names to have "Type" suffix

Flow has a similar keyword "**interface**".  
For the difference, see <https://stackoverflow.com/questions/43023941/flow-interfaces-versus-types>



# Functions

- Can specify parameter and return types

`ResultType` is an object type defined elsewhere.

```
function monopoly(passGo: boolean, dice: number, piece: string): ResultType { ... }
```

- Can use in type aliases

- useful when passing functions to others and returning functions from them

```
type MyFnType = (passGo: boolean, dice: number, piece: string) => ResultType;
```

- note => instead of : before return type

- Add ? after names of optional parameters

```
type DistanceFromOriginFnType = (x: number, y: number, z?: number) => number;
```

- Add ? before parameter types where `null` and `undefined` are allowed

```
type CallbackType = (err: ?Error, result?: mixed) => void;
```

- Polymorphic functions (parametric types)

```
type MyFnType<T> = (p1: T, p2: T) => T;
```

more on this later  
on "Generics" slide

- can have any number of parametric types, but more than two is unusual

# Function Examples

function.js

- Note how functions that are missing ending parameters and/or the return type are considered to match

```
// @flow

type TestFnType =
  (p1: boolean, p2: number, p3: string) => void;

// Perfect match
const f1: TestFnType =
  (a: boolean, b: number, c: string): void => {};

// Missing last parameter and return type - OK
const f2: TestFnType = (a: boolean, b: number) => {};

// Missing last two parameters and return type - OK
const f3: TestFnType = (a: boolean) => {};

// Missing all parameters and return type - OK
const f4: TestFnType = () => {};

// Wrong type for first parameter - error
const f5: TestFnType = (a: number) => {};

// Missing return type and returns wrong type - error
const f6: TestFnType =
  (a: boolean, b: number, c: string) => 'bad';

// Wrong return type - error
const f7: TestFnType =
  (a: boolean, b: number, c: string): string => 'bad';
```



# Arrays

- `Array<element-type>` or `element-type[]`
  - ex. `Array<Date>` or `Date[]`
- Can nest these
  - ex. `Array<Array<number>>` or `number[][]`

```
type ArrayOfArraysOfNumbersType = Array<Array<number>>;  
const aoaon: ArrayOfArraysOfNumbersType = [[1, 2], [3, 4, 5]];
```

array-of-array.js

- Tuples
  - fixed-size arrays where elements at specific indexes have specific types
  - not necessarily the same type at every index
  - ex. `type PointType = [number, number];`
  - can set value at a specific index
  - cannot use `Array` methods that mutate it

```
// @flow

type PointType = [number, number]; // a tuple
type PointArrType = Array<PointType>;

function distance(p1: PointType, p2: PointType): number {
  return Math.hypot(p2[0] - p1[0], p2[1] - p1[1]);
}

function perimeter(points: PointArrType): number {
  return points.reduce(
    (sum: number, point: PointType, index: number) =>
      sum += index ?
        distance(points[index - 1], point) :
        distance(points[points.length - 1], point),
    0);
}

const points: PointArrType = [
  [0, 0], [3, 4], [5, 2]
];

console.log('perimeter =', perimeter(points).toFixed(2)); // good
console.log('perimeter =', perimeter(7)); // error
```

It may be more common to use an object or class with *x* & *y* properties.

previous point to current  
last point to first

```
23: console.log('perimeter =', perimeter(7));
                                ^ number. This type is incompatible with the expected param type of
10: function perimeter(points: PointArrType): number {
    ^^^^^^^^^^^^^ array type

Found 1 error
```



- 1) Object or {}
  - can have any properties
- 2) Object signature
  - list of properties and their types
  - required by default;  
follow names of optional properties with ?
  - actual objects can have additional properties
  - to disallow additional properties, use  
{ | ... | } or \$Exact<{ ... }>
- 3) class or constructor function name
  - specifies type of objects allowed
  - can be a built-in or custom class
    - ex. Array, Date, Error, Map, RegExp, and Set

```
// @flow
type PersonType = {
  name: string,
  birthday: Date,
  spouse?: ?PersonType
};
```

height was purposely omitted to show that objects of this type can have additional properties

add ? before types where null or undefined is allowed

```
const tami: PersonType = {
  name: 'Tami',
  birthday: new Date(1961, 8, 9),
  height: 65
};

const mark: PersonType = {
  name: 'Mark',
  birthday: new Date(1961, 3, 16),
  height: 74,
  spouse: tami
};
```

# ... Objects

plain-object.js

```
// @flow

function logProps(obj: Object) {
  Object.keys(obj).forEach(key =>
    console.log(key, '=', obj[key]));
}

logProps({foo: 1, bar: 2}); // good
logProps(7); // error
```

```
8: logProps(7);
      ^ number. This type is incompatible with the expected param type of
3: function logProps(obj: Object) {
      ^^^^^^ object type
```



- Can have any string keys and all values are often the same type

just serves as  
a description  
of the keys

```
// @flow

type PlayerToNumberMapType = {
  [player: string]: number
};

const playerToNumberMap: PlayerToNumberMapType = {
  'Mario Lemieux': 66,
  'Wayne Gretzky': 99
};

Object.keys(playerToNumberMap).forEach(player => {
  const number = playerToNumberMap[player];
  console.log(`${player} is number ${number}`);
});
```

# Classes

class.js

```
// @flow
class Person {
  name: string;
  birthday: Date;
  height: number;
  spouse: Person;

  constructor(name: string, birthday: Date, height: number): void {
    this.name = name;
    this.birthday = birthday;
    this.height = height;
  }

  marry(person: Person): void {
    this.spouse = person;
    person.spouse = this;
  }
}

const tami: Person = new Person('Tami', new Date(1961, 8, 9), 65);
const mark: Person = new Person('Mark', new Date(1961, 3, 16), 74);
tami.marry(mark);

function logPerson(person: Person): void {
  const status: string = person.spouse ?
    'married to ' + person.spouse.name : 'single';
  console.log(person.name + ' is ' + status + '.');
}

logPerson(mark); // good
logPerson(new Date()); // error
```

declarations of  
properties that are  
in each instance

Values for class properties are  
optional and there isn't a way  
to make them required.

```
31: logPerson(new Date());
           ^^^^^^^^^^^ Date. This type is incompatible with
25: function logPerson(person: Person) {
           ^^^^^^ Person
```



# Class Types

**Advanced Feature**

class-type.js

- A type can refer to a class rather than an instance of it
- To declare these, use `Class<Class-Name>`
- Variables of the type hold a reference to a class and can be used to create instances of it
- Example

```
// @flow

// These classes could be more involved.
class Animal {}
class Mineral {}
class Vegetable {}

type AMVType = Animal | Mineral | Vegetable;

const alive = false;
const grows = false;
const clazz: Class<AMVType> = alive ? Animal : grows ? Vegetable : Mineral;

const thing: AMVType = new clazz();
console.log(thing instanceof Mineral); // true
```

No type errors!

# Generics

a.k.a. parameterized types

## Advanced Feature

- Four uses
  - class definitions
  - interface definitions
  - function definitions
  - type aliases for objects
- Typical name for generic type is `T`, but any name works
- Can have more than one, separated by commas

```
class MyClass<T> {
  // use T in the definitions of
  // properties and/or methods
}

function myFunction<T>() {
  // use T in types of parameters,
  // return type, and variables
}
```

generics.js

```
type PricedType<T> = {
  item: T, price: number, date: Date
};

function logPricedType<T>(priced: PricedType<T>) {
  console.log(String(priced.item),
    'cost', priced.price,
    'on', priced.date.toString());
}

const apple: PricedType<string> = {
  item: 'Gala apple',
  price: 0.99,
  date: new Date()
};
logPricedType(apple); // Gala apple cost 0.99 on Fri Apr 07 2017\

class Fruit {
  kind: string;
  constructor(kind: string): void {
    this.kind = kind;
  }
  toString(): string {
    return this.kind;
  }
}

const banana: PricedType<Fruit> = {
  item: new Fruit('Chiquita banana'),
  price: 0.29,
  date: new Date()
};
logPricedType(banana); // Chiquita banana cost 0.29 on Fri Apr 07 2017\
```

No type errors!



# Maps and Sets

```
// @flow

type PlayerType = {name: string, number: number, position: string};

const gretzky: PlayerType =
  {name: 'Wayne Gretzky', number: 99, position: 'center'};
const lemieux: PlayerType =
  {name: 'Mario Lemieux', number: 66, position: 'center'};

const players: PlayerType[] = [gretzky, lemieux];
const playerSet: Set<PlayerType> = new Set(players);
const playerMap: Map<number, PlayerType> = new Map();

for (const player of players) {
  playerMap.set(player.number, player);
}

console.log('map-set.js: playerSet =', playerSet);
console.log('map-set.js: playerMap =', playerMap);
```

```
map-set.js: playerSet = Set {
  { name: 'Wayne Gretzky', number: 99, position: 'center' },
  { name: 'Mario Lemieux', number: 66, position: 'center' } }
map-set.js: playerMap = Map {
  99 => { name: 'Wayne Gretzky', number: 99, position: 'center' },
  66 => { name: 'Mario Lemieux', number: 66, position: 'center' } }
```

- Used to describe commonality between classes that isn't expressed through a common superclass

If you have the ability to modify the classes, it may be best to give them a common superclass.

```
interface Vehicle {
  start(): void,
  stop(): void
}
class Boat implements Vehicle {
  start(): void { console.log('The boat is started.')}
  stop(): void { console.log('The boat is stopped.')}
}
class Car implements Vehicle {
  start(): void { console.log('The car is started.')}
  stop(): void { console.log('The car is stopped.')}
}
class House {} // has no methods
function testDrive(vehicle: Vehicle) {
  vehicle.start();
  vehicle.stop();
}
const boat: Vehicle = new Boat();
testDrive(boat); // good
const car: Vehicle = new Car();
testDrive(car); // good
const house: Vehicle = new House();
testDrive(house); // error, not a Vehicle
```

can describe properties and methods

optional

A class can implement multiple interfaces, separated by commas.

```
38: const house:Vehicle = new House();
      ^^^^^^^ property `start` of Vehicle.
      Property not found in
38: const house:Vehicle = new House();
      ^^^^^^^^^^^^^ House

38: const house:Vehicle = new House();
      ^^^^^^^ property `stop` of Vehicle.
      Property not found in
38: const house:Vehicle = new House();
      ^^^^^^^^^^^^^ House
```

Found 2 errors

A House doesn't have the properties of a Vehicle.



- Can declares that a value can have one of a list of types

Also see “intersection types”, but those don’t seem very useful. They require a value to conform to multiple types.

ThingType is a “**disjoint union**”. Each type in the union is an object that is distinguished from the others based on a property with a specific value (**type** in this case).

```
type PrimitiveType = boolean | number | string;
let value: PrimitiveType = true;
value = 7; // good
value = 'foo'; // good
value = {}; // error

type AnimalType = {name: string, type: 'animal'};
type MineralType = {name: string, type: 'mineral'};
type VegetableType = {name: string, type: 'vegetable'};
type ThingType = AnimalType | MineralType | VegetableType;

const dog: AnimalType = {name: 'Dasher', type: 'animal'};
const mineral: MineralType = {name: 'amethyst', type: 'mineral'};
const vegetable: VegetableType = {name: 'corn', type: 'vegetable'};

let thing: ThingType = dog; // good
console.log(thing.name); // Dasher
thing = mineral; // good
console.log(thing.name); // amethyst
thing = vegetable; // good
console.log(thing.name); // corn
thing = {name: 'bad', type: 'other'}; // error
```

Functions that have parameters of a union type must handle all possible types

# Unions For Enums

enum-union.js

- Can declare that a value can be one of a list of literal value

```
type ActivityType = 'swim' | 'bike' | 'run';

let activity: ActivityType = 'swim'; // good
console.log('Your current activity is', activity);

activity = 'bike'; // good
console.log('Your current activity is', activity);

activity = 'run'; // good
console.log('Your current activity is', activity);

activity = 'collapse'; // error
```

- To define an enum type whose values come from the keys of an object at runtime

```
type MyType = $Keys<typeof myObj>;
```



# Sharing Type Aliases

- Allows them to be used in many source files
- Export from files where they are defined with `"export type"`
- Import to use in other files with `"import type"`
- Requires use of a module bundler like webpack
- For more information, see <http://flowtype.org/docs/modules.html#type-imports-exports>

# Sharing Type Aliases Example

```
// @flow                                     type-alias-export.js

// This type matches any object that has
// a name property with a type of string.
export type NamedType = {name: string};

export function sayHello(thing: NamedType): void {
  console.log('Hello, ' + thing.name + '!');
}
```

```
// @flow                                     type-alias-import.js

import type {NamedType} from './type-alias-export';
import {sayHello} from './type-alias-export';

const mark: NamedType = {name: 'Mark', hobby: 'running'};
sayHello(mark); // good
sayHello({name: 'Tami', hobby: 'swimming'}); // good
sayHello('Mark'); // error
```

```
9: sayHello('Mark');
   ^^^^^^ string. This type is incompatible with the expected param type of
5: export function sayHello(thing: NamedType) {
   ^^^^^^^^^ object type. See: type-alias-export.js:5
```



# Escape Hatch

- Sometimes, not often, Flow can't be easily satisfied
- To disable Flow type checking for a single line, precede the line with this comment

```
// $FlowFixMe optional description of why
```

- Examples
  - destructuring from an object with unknown properties?
  - more? - LOOK FOR EXAMPLES IN LAUNCHPAD CODE



# Flow Server ...

- The brains behind Flow
- Analyzes and stores many things about the flow of code in an application
  - variable/function types, locations of their definitions, references to them, ...
  - performs parallel evaluation of multiple files in the background for performance
    - starting the flow server starts several "flow" processes, based on # of cores and `server.max_workers` option
- Considers all `.js` files under directory containing `.flowconfig`
  - unless specified differently in `.flowconfig` file
- Initially checks all files, then only checks ...
  - files that have changed
  - files that import from files that have changed
  - newly created files
- Doesn't output error messages, just collects them
  - queried by Flow CLI, editors, and IDEs

creating this file  
is discussed later



# ... Flow Server

- To start server
  - in background, `flow start`
  - in foreground, `flow server`
- To output errors collected by server
  - `flow status` or just `flow`
  - if server is not running, these start it and run a full check
- To check all files and output errors
  - `flow check`
  - if server is not running, this starts one, performs checking, and stops it
- To stop server
  - `flow stop` from directory where it is running

The server must be stopped and restarted after changes to `.flowconfig` or declaration files because the server caches those.

# Flow CLI

- Provides CLI commands that can be used by editors/IDEs to obtain information it has collected
- Many commands take a file path, line number, and column number (referred to here as “FLC”) to identify a variable/function of interest and output a list of the same
  - column number can be any column within the name of a variable or function, not just beginning



# Flow CLI Commands

- Commands that query the server are intended to be used from editor plugins
- They include
  - **autocomplete** - inserts "magic autocomplete token" at specified position; unclear how to use
  - **coverage** - outputs percentage of expressions in a given file for which Flow knows their types
  - **find-refs** - outputs FLCs that refer to the variable at an FLC
    - `flow find-refs some-name.js line-num col-num`
  - **gen-flow-files** - generates a `.js.flow` file containing type declarations from a `.js` file
    - `flow gen-flow-files some-name.js > some-name.js.flow`
  - **get-def** - gets FLCs for beginning and end of name in the definition of a variable at an FLC
    - `flow get-def some-name.js line-num col-num`
  - **get-imports** - gets FLCs of each imported module in a given file
    - `flow get-imports some-name.js`
  - **type-at-pos** - gets type of variable at FLC and FLCs for all references to it
    - `flow type-at-pos some-name.js line-num col-num`
  - may support refactorings in the future

To get help on a specific command,  
`flow command --help`



# Flow Project Checklist

- 1) Install “dev dependencies”
- 2) Add npm scripts to `package.json`
- 3) Setup ESLint
- 4) Setup Babel
- 5) Create `.flowconfig` file
- 6) Use `flow-typed` to get dependency type declarations
- 7) Configure editors/IDEs to use Flow



# Browser vs. Server Apps

- The example that follows is a Node.js project that uses CommonJS modules
- A web UI project that uses ES Modules would need a module bundler like **Webpack** or **Rollup**
  - configuring that is beyond the scope here
  - if targeting React, consider using **create-react-app** which configures Webpack and much more for you



# Dev Dependencies

- `cd` to top project directory containing `package.json`
  - if that file doesn't exist yet, enter `npm init` to create it
- For each of these, enter `npm install -D name` or `yarn add -D name`
  - **babel-cli** - command-line interface to Babel transpiler
  - ✓ • **babel-eslint** - alternate parser for ESLint that understands ES6+ syntax
  - ✓ • **babel-plugin-transform-flow-strip-types** - removes Flow type annotations from Babel output
  - ✓ • **babel-preset-env** - automatically determines needed Babel plugins and polyfills based on target environment
  - ✓ • **eslint** - JavaScript linter
  - ✓ • **eslint-plugin-flowtype** - implements ESLint rules to check usage of Flow types
    - **flow-bin** - the Flow type checker
    - **flow-watch** - "file watcher that clears the console (terminal) and runs flow on each change"
    - **npm-run-all** - "run multiple npm-scripts in parallel or sequentially"

**create-react-app  
provides most of  
this setup for you!**

✓ = installed by create-react-app



# Recommended npm Scripts

- **"babel": "babel src -d build"**
  - transpiles all `.js` files under `src` directory into `build` directory
- **"flow": "flow"**
  - runs `flow` on all `.js` files in the project or only those specified in `.flowconfig`
- **"floww": "flow-watch"** not needed if editor/IDE does this
  - same as `flow`, but keeps running, watching files for changes
- **"lint": "eslint --quiet src"** --quiet only reports errors
  - runs `eslint` on all `.js` files under `src` directory
- **"run": "node build/index.js"**
  - runs transpiled version of application
- **"start": "npm-run-all lint flow babel run"**
  - combines previous steps

# ESLint Setup

modify these settings  
based on personal preference

```
{  
  ...  
  "parser": "babel-eslint",  
  ...  
  "plugins": [  
    "flowtype"  
  ],  
  ...  
  "rules": {  
    _____  
  }  
}
```

**.eslintrc.json**

```
...  
"flowtype/boolean-style": ["error", "boolean"],  
"flowtype/define-flow-type": ["error", {"no-undef": "error"}],  
"flowtype/delimiter-dangle": ["error", "never"],  
"flowtype/generic-spacing": ["error", "never"],  
"flowtype/no-dupe-keys": "error",  
"flowtype/no-primitive-constructor-types": "error",  
"flowtype/no-weak-types": "warn",  
"flowtype/object-type-delimiter": ["error", "comma"],  
"flowtype/require-parameter-type": "off",  
"flowtype/require-return-type": "off",  
"flowtype/require-valid-file-annotation": "off",  
"flowtype/semi": ["error", "always"],  
"flowtype/sort-keys": "off",  
"flowtype/space-after-type-colon": ["error", "always"],  
"flowtype/space-before-generic-bracket": ["error", "never"],  
"flowtype/space-before-type-colon": ["error", "never"],  
"flowtype/type-id-match": "error",  
"flowtype/union-intersection-spacing": ["error", "always"],  
"flowtype/use-flow-type": "error",  
"flowtype/valid-syntax": "error",  
...
```

For rule details, see  
<https://github.com/gajus/eslint-plugin-flowtype#eslint-plugin-flowtype-rules>



# Babel Setup

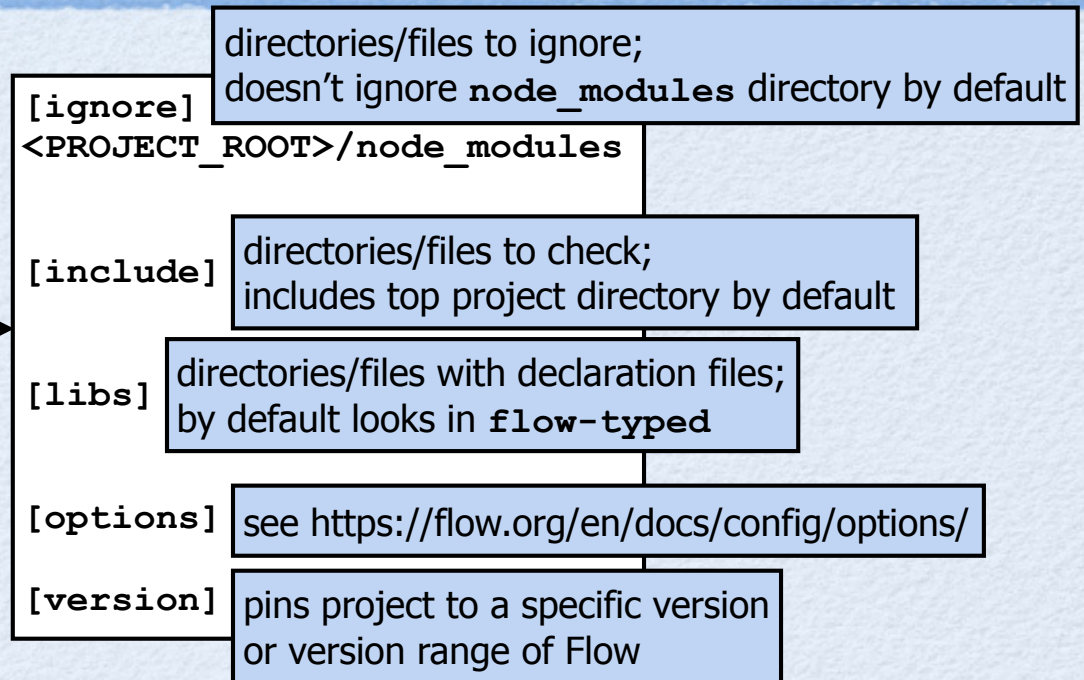
```
{  
  "presets": [  
    ["env", {  
      "targets": {  
        "node": 7.7  
      }  
    }]  
  ],  
  "plugins": [  
    "transform-flow-strip-types"  
  ]  
}
```

`.babelrc`

change for a different Node version or a web app

# .flowconfig File

- cd to top project directory
- Run "flow init"
- Creates .flowconfig file
  - with all sections empty
- Comment lines start with # or ; optionally preceded by whitespace
- Ignores are processed after includes
- For more information, see <https://flow.org/en/docs/config/>



It is **essential** to have a .flowconfig file in the project root directory! Flow searches upward until this file is found. If it reaches the top directory without finding one, Flow will check every JavaScript file below that, which **will heavily tax your computer**. If this happens, **kill the Flow server** by entering `flow stop` or `killall flow` (on \*nix systems).



# .flowconfig Options

- **module.file\_ext=***file-extension*
  - by default, checks `.js`, `.jsx`, and `.json` files
  - specify one or more of these to change
- **all=true**
  - checks all files with certain file extensions, not just those annotated with `// @flow`
- **emoji=true**
  - adds emoji to Flow status messages output when it checks files
- **munge\_underscores=true**
  - treats underscore-prefixed class properties and methods as private
- and many more



# flow-typed

- “A central repository for Flow library definitions”

- <https://github.com/flowtype/flow-typed>

- Steps to use in a project

- 1) `npm install -g flow-typed`
  - 2) cd to project directory
  - 3) `flow-typed install`
    - creates `flow-typed` directory if not present
    - installs type declaration files there for all dependencies found in `package.json`
    - generates “stubs” for dependencies that do not yet have type declaration files in the `flow-typed` repository
  - 4) add `flow-typed` directory to version control
  - to update previously installed type definitions  
`flow-typed update`

**Examples** include Axios, Chalk, Enzyme, Express, Jasmine, Jest, Lodash, Moment, pg (Postgresql library), react-redux, React Router, Redux, and RxJS.



# flow-typed install

- Generates type definitions for dependencies in `package.json`

```
🏃 flow-typed install
• Found flow-bin@v0.42.0 installed. Installing libdefs compatible with this version of Flow...
• Found 10 dependencies in package.json. Searching for libdefs...
• rebasing flow-typed cache...done.
• Installing 2 libdefs...
  • lodash_v4.x.x.js
    ↳ ./flow-typed/npm/lodash_v4.x.x.js
  • flow-bin_v0.x.x.js
    ↳ ./flow-typed/npm/flow-bin_v0.x.x.js
• Generating stubs for untyped dependencies...
  • liner@^0.3.3
    ↳ flow-typed/npm/liner_vx.x.x.js
  • babel-eslint@^7.2.1
    ↳ flow-typed/npm/babel-eslint_vx.x.x.js
  • babel-preset-env@^1.2.2
    ↳ flow-typed/npm/babel-preset-env_vx.x.x.js
  • babel-plugin-transform-flow-strip-types@^6.22.0
    ↳ flow-typed/npm/babel-plugin-transform-flow-strip-types_vx.x.x.js
  • babel-cli@^6.24.0
    ↳ flow-typed/npm/babel-cli_vx.x.x.js
  • npm-run-all@^4.0.2
    ↳ flow-typed/npm/npm-run-all_vx.x.x.js
  • eslint@^3.18.0
    ↳ flow-typed/npm/eslint_vx.x.x.js
  • eslint-plugin-flowtype@^2.30.4
    ↳ flow-typed/npm/eslint-plugin-flowtype_vx.x.x.js

!! No flow@v0.42.0-compatible libdefs found in flow-typed for the above untyped dependencies !!

I've generated `any`-typed stubs for these packages, but consider submitting
libdefs for them to https://github.com/flowtype/flow-typed/
```

If this prompts for your **GitHub username**  
**and password**, just **press enter** for both.  
See <https://github.com/flowtype/flow-typed/721>.

To update previously  
installed type definitions,  
enter **flow-typed update**

# flow-typed Results

- `flow-typed/npm/lodash_v4.x.x.js`
  - `startsWith(string?: string, target?: string, position?: number): bool;`
- `flow-typed/npm/liner_vx.x.x.js`
  - auto-generated since this isn't a popular package
  - uses `any` type for everything



- **Atom**

- search for these at <https://atom.io/packages>:  
flow-ide, Nuclide (from Facebook), linter-flow, autocomplete-flow
  - the package "flow" is for the haxe flow build tool, not Facebook Flow

You might also like  
vim-mode-plus.

- **emacs**

- <https://github.com/flowtype/flow-for-emacs>
- <https://github.com/lbolla/emacs-flycheck-flow>

- **Sublime**

- <https://github.com/SublimeLinter/SublimeLinter-flow>

- **Visual Studio Code**

- search for these at <https://marketplace.visualstudio.com/>:  
"Flow Language Support" and vscode-flow-ide

Disable default syntax validation in TypeScript section:  
`"javascript.validate.enable": false`  
This runs Flow on file saves.  
Can hover over a variable or  
function name to see its type.

You might also  
like VsCodeVim.  
Press "Reload"  
button to enable.

- **WebStorm**

- see <https://blog.jetbrains.com/webstorm/2016/11/using-flow-in-webstorm/>  
and <https://www.jetbrains.com/help/webstorm/2017.1/flow-type-checker.html>



# ... Editor/IDE Setup

- **Vim**

It is beneficial to use both options.

- **option #1** - Asynchronous Linting Environment (**ALE**)

- <https://github.com/w0rp/ale>
- integrates with a large number of linters for many syntaxes
- make sure 'flow' is one of the enabled linters for JavaScript

In .vimrc

```
let g:ale_linters = {  
  \ 'javascript': ['eslint', 'flow'],  
  \  
}
```

- **option #2** - **vim-flow** plugin

- <https://github.com/flowtype/vim-flow>
- adds object property and method completions using "Omni completion" which must be enabled
  - to trigger, press c-x c-o
  - move up and down in list of completions with tab/shift-tab, c-n/c-p, or down/up arrows
  - continue typing to use selection
- commands
  - **:FlowMake** - runs Flow on all files in project with @flow annotation and displays results in quickfix window
  - **:FlowToggle** - toggles type checking on save
  - **:FlowType** - displays type of variable under cursor (I mapped to <leader>ft)
  - **:FlowJumpToDef** - jumps to definition of variable under cursor (I mapped to <leader>fj)

**ALE plugin** starts a flow server when the first JavaScript file is opened. Exiting Vim does not stop it. Entering "flow stop" also doesn't stop it. Enter "killall flow" to stop it.

The first time one of these commands is run, it may take about 10 seconds for a Flow server to be started, and run the initial check.



# Sample Project

- Uses a popular npm package (**lodash**) and an unpopular one (**liner**)
- Reads lines from a text file using **liner**, uses the **lodash** function **startCase** to capitalize each word, outputs each line, and outputs the number of lines read
- To run **ESLint**, **Flow**, **Babel**, and the application, enter `"npm start"`
  - assumes npm scripts described earlier

# Project Dependencies

- `cd` to top project directory containing `package.json`
- For each of these, enter  
`"npm install -S name"` or `"yarn add name"`
  - **liner** - "reads lines from files and streams"
  - **lodash** - "modern JavaScript utility library delivering modularity, performance & extras"



writes each line  
in the file to  
the console

```
// @flow
import filer from './filer';

filer('./haiku.txt', (lineCount: number) => {
  console.log('line count is', lineCount);
});
```

index.js

Out of memory.  
We wish to hold the whole sky,  
But we never will.

haiku.txt

note the  
case changes

Out Of Memory  
We Wish To Hold The Whole Sky  
But We Never Will  
line count is 3

Output

# ... The Code

```
// @flow
const Liner = require('liner');
const _ = require('lodash/string');

/**
 * Outputs each line in the text file at the given path,
 * capitalizing the first letter of each word,
 * and calls cb with the number of lines read.
 */
function processFile(path: string, cb: (number) => void): void {
  let count = 0;
  const liner = new Liner(path);

  liner.on('readable', () => {
    while (true) {
      const line = liner.read();
      if (line === null) break;
      console.log(_.startCase(line));
      count++;
    }
  });

  liner.on('end', () => cb(count));

  liner.on('error', err => console.error(err));
}

export default processFile;
```

filer.js



# React/JSX Support ...

- React supports two ways of defining components, class-based and stateless functional
- In stateless functional components, Flow can be used to specify the types of props obtained through destructuring of the props object
- In class-based components, Flow can be used to specify the types of props, default props, and state
- For more details, see <https://flowtype.org/docs/react.html>

# Type of `this.props` ...

- In a **functional component**,  
declare type of this object, not types within destructuring

- this does not work

```
const MyComponent = ({foo: string, bar: number}) => { ... };
```

- this does work

```
const MyComponent = ({foo, bar}: {foo: string, bar: number}) => { ... };
```

- better yet

```
type PropsType = {foo: string, bar: number};
const MyComponent = ({foo, bar}: PropsType) => { ... };
```



# ... Type of `this.props`

- In a **class component**, declare props, state, and methods as “public class fields”

```
// @flow
import React from 'react';
type PropsType = {foo: string, bar: number};
class MyComponent extends React.Component {
  props: PropsType;
  handleClick: () => void;

  handleClick = () => console.log('got click');

  render() {
    const {bar, foo} = this.props;
    return (
      <div onClick={this.handleClick}>
        <div>foo = {foo}</div>
        <div>bar = {bar}</div>
      </div>
    );
  }
}

export default MyComponent;
```

could declare state the same way ...  
state: StateType;

# Built-in React Types

- See <https://github.com/facebook/flow/blob/master/lib/react.js>
- **Component** class - includes lifecycle methods
- **Element** class - instances are typically created using JSX
- **PropTypes** object type - for declaring types of component props in the React way
  - includes `any`, `array`, `arrayOf`, `bool`, `element`, `func`, `instanceof`, `node`, `number`, `object`, `objectOf`, `oneOf`, `oneOfType`, `shape`, and `string`
- **react** module
  - includes `createElement`, `renderToString`, and more
- **react-dom** module
  - includes `findDOMNode` and more
- **Events**
  - **SyntheticEvent** and subclasses including **SyntheticDragEvent**, **SyntheticInputEvent**, **SyntheticKeyboardEvent**, **SyntheticMouseEvent**, **SyntheticTouchEvent**, **SyntheticWheelEvent**, and more

These are the actual types of the event objects used by React. They contain a `nativeEvent` property for accessing the DOM event object.



# Flow vs. PropTypes

- There are benefits to using Flow types in place of React PropTypes, but there are also downsides
- **Flow types** allow type errors in props to be detected in Flow-aware editors
- **Flow** can also be run as part of the build process and a build can be aborted if any errors are found
- **React PropTypes** allow errors to be flagged in tests and when the app is run by displaying messages in the browser console
- Can specify types for props **using both** Flow types and React PropTypes to get both sets of benefits
- React developers may gradually shift to only using Flow types for props
  - it is tedious to specify types of props in two ways and keep them in sync when changes are needed



# Library Definitions

- Declare types of globals (types, variables, functions, and classes) and modules (CommonJS or ES) without modifying the code where they are defined
  - used when definitions cannot be modified to add type declarations
  - for example, libraries like lodash
- Allows usages to be type-checked
- For details, see <https://flow.org/en/docs/libdefs/creation/>



# Locations For Types

- In source files
  - what we have seen so far
  - preferred when source files can be modified
- In **flow-typed** directory
  - installed using "**flow-typed install**" command described next
  - preferred when source files cannot be modified
- In files in directories listed in **.flowconfig** under "**include**"
- In other files in same directory with same name, but with **.flow** appended
  - ex `foo.js` and `foo.js.flow`

# Kinds of Declarations

This is Flow-specific syntax,  
not JavaScript!

- Variable

```
declare var name: type;
```

- Function

```
declare function name(p1-name: p1-type, ...) return-type;
```

- Class / Interface

```
declare [class|interface] class-name {  
  constructor(p1-name: p1-type): class-name;  
  static method-name(p1-name: p1-type, ...): return-type;  
  method-name(p1-name: p1-type, ...): return-type;  
}
```

- Flow Type

```
declare type name = type;
```

- Modules

- see next slide



# Modules

- Provide a named scope for variables, functions, classes, types, and interfaces
- Otherwise those are global
- Defined in a file with the same name as module they describe
- Uses Flow-specific syntax, not JavaScript!
- Reference in other files using ***module-name.thing-name***
- For more detail, see <https://flow.org/en/docs/config/>

# CommonJS Module Example

math.js

```
function double(n) {  
  return n * 2;  
}  
  
exports.double = double;
```

implementation file

demo.js

```
// @flow  
const math = require('./math');  
  
console.log(math.double(3)); // good  
console.log(math.double('bad')); // error
```

math.js

```
declare module './math' {  
  # type declarations go here  
  declare module.exports: {  
    double(n: number): number;  
  }  
}
```

type declaration file



# ES Module Example

math.js

```
export function double(n) {  
  return n * 2;  
}
```

implementation file

demo.js

```
// @flow  
import * as math from './math';  
  
console.log(math.double(3)); // good  
console.log(math.double('bad')); // error
```

math.js

```
declare module './math' {  
  # type declarations go here  
  declare export function double(n: number): number;  
}
```

type declaration file

# .js.flow Declarations

- Colocated with implementation file
  - ex. `math.js` and `math.js.flow`
- Declares types for anything that is exported
- Types declared in `.js.flow` file are used in place of those in `.js` file, if any
  - useful when `.js` file doesn't include type annotations or they are incorrect and cannot be modified
  - supports keeping `.js` files free of Flow-specific syntax
  - allows them to be used without tooling to strip out type annotations

```
// @flow
```

`math.js.flow`

```
declare export function double(n: number): number;
```

This is Flow-specific syntax,  
not JavaScript!



# Summary

- Do the benefits derived from types justify the extra work required to specify them?
- In my experience with Flow I have been surprised at how often adding types uncovered issues in existing code
- I highly recommend giving Flow, or TypeScript, a try!
- Start simple, perhaps just adding types for function parameters and return types
- Over time your confidence in the quality of the code will increase and the number of errors you discover at runtime will decrease!

# Resources

- Main site
  - <https://flow.org/>
- “Flow type cheat sheet”
  - <http://www.saltycrane.com/blog/2016/06/flow-type-cheat-sheet/>



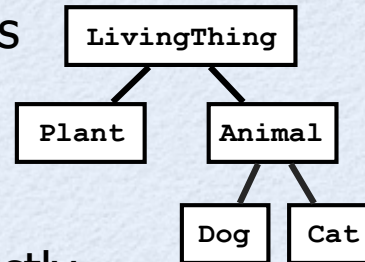
# Nominal vs. Structural Typing

- Nominal typing
  - determines whether values are compatible based on the name of their type
  - ex. objects from `Plant` class are not compatible with objects from `Dog` class
- Structural typing
  - determines whether values are compatible by their set of allowed values
  - in the case of objects this includes properties and methods
  - objects from two classes are compatible if they have the same properties and methods
- TypeScript
  - only uses structural typing
- Flow
  - functions and literal objects are structurally typed
  - objects from classes are nominally typed
  - can get structural typing between objects from classes by using interfaces
  - see <https://flow.org/en/docs/lang/nominal-structural/>



# Variance

- Consider a hierarchy of living things



- Invariant** - types must match exactly
  - if `Dog` is allowed, only `Dog` can be used
- Covariant** - can use subtype in place of supertype
  - can use `Dog` anywhere `LivingThing` is allowed
- Contravariant** - can use supertype in place of subtype
  - can use `LivingThing` anywhere `Dog` is allowed



# Variance in Flow & TypeScript

```
// @flow

class LivingThing {}
class Plant extends LivingThing {}
class Animal extends LivingThing {}
class Dog extends Animal {}
class Cat extends Animal {}

const animals: Animal[] = [];
// In Flow, adding elements to arrays is covariant (can add subtypes).
animals.push(new Dog());
animals.push(new Cat());
animals.push(new Plant()); // error in Flow, but not TS

// In Flow, extracting elements from arrays is contravariant
// (get type of array which is the supertype of the elements)
const dog: Dog = animals[0]; // error in Flow, but not TS
//const dog: Animal = animals[0]; // works
console.log('dog =', dog);

const plant: Plant = animals[0]; // error in Flow, but not TS
console.log('plant =', plant);
```

**TypeScript** relies on structural typing rather than nominal. It views all these classes as equivalent because they have the same properties and methods.

# Soundness vs. Correctness

- **Soundness**
  - “ability for a type checker to catch every single error that might happen at runtime”
  - “comes at the **cost** of sometimes **catching errors that will not actually happen** at runtime”
- **Completeness**
  - “ability for a type checker to only ever catch errors that would happen at runtime”
  - “comes at the **cost** of sometimes **missing errors** that will happen at runtime”
- Ideally want both
  - to catch every error that **will** happen at runtime
  - but this is not possible in JavaScript
- Flow
  - “tries to be as sound and complete as possible ... but ... has to make a tradeoff”
  - “tends to favor soundness over completeness, ensuring that code doesn’t have any bugs”
- TypeScript
  - favors completeness, only reporting real errors and possibly missing errors



# Sealed Objects

- Empty object literals are not sealed
  - can add properties later with values of any type
- Object literals with at least one property are sealed
  - cannot add properties later
  - can change existing property values, but their types are inferred from initial values and non-compatible values cannot be assigned later

```
// @flow

// Sealed object
const obj1 = {foo: 1};
obj1.bar = 2; // error
obj1.foo = 2; // okay
obj1.foo = 'test'; // error

// Unsealed object
const obj2 = {};
obj2.foo = 1; // okay
obj2.foo = 'test'; // okay
```