JSDoc*, JSIDL*, Code Editors
Static Analysis of JavaScript

1. Tokenize and Parse
2. ???
3. Profit (Program Understanding)
Develop-time uses of Type Inferencing

- Content Assist / Code Completion
- Code Validation
- Jump to Declaration / References
- Refactoring
- Source Templates
Type Inferencing is a Hard Problem

What we get for free…

• Grammar and Keywords
• Scope Analysis
• Flow Analysis

• Dynamic Code Analysis, Code Heuristics
  – Interesting but increasingly imprecise
(JSDoc*) - We Need Help from the User

```javascript
/**
 * @typedef Person {firstname: string, lastname: string}
 *
 */

/**
 * @param {Person} person
 *
 */
function greeter(person) {
    return "Hello, " + person.firstname + " " + person.lastname;
}

/** @type Person */
var user = {firstname: "Jane", lastname: "User");

document.body.innerHTML = greeter(user);
```

- Comment-based
- Descriptive tag (@tagName)
- Text and Location Semantics
Interface Person {
    firstname: string;
    lastname: string;
}

function greeter(person: Person) {
    return "Hello, " + person.firstname + " " + person.lastname;
}

var user = {Firstname: "Jane", lastname: "User"};
document.body.innerHTML = greeter(user);

/**
 * @typedef Person {firstname: string, lastname: string}
 */

/**
 * @param {Person} person
 */
function greeter(person) {
    return "Hello, " + person.firstname + " " + person.lastname;
}

/*@type Person*/
var user = {firstname: "Jane", lastname: "User"};
document.body.innerHTML = greeter(user);
Standardize JSDoc?

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

SITUATION: THERE ARE 14 COMPETING STANDARDS.

14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES. YEAH!

SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.
JSDoc Proposal

- All JSDoc groups do not currently support ES6 or even ES5 particularly well
- Socialize idea of common work
- Work with an existing JSDoc definition and set of adopters
- Standardize meaning of tags, tag content and location semantics
JS IDL?

- Representation of the public interface of a software component / artifact

- viewed in the opposite direction is this the product of type inferencing? (e.g. a type definition)
Type definition...

- TypeScript Definition
  - (Definitely Typed)

- Tern style JSON definition
  - (Supports Type Expressions)(Tern, Orion)

- JSDoc mixed with simple JS definition
  - (common a few years ago)

---

- Green field approaches
  - WebIDL        --> JS IDL
  - JS           --> JS IDL
declare module Ember {

    /**
     * Alias for jQuery.
     */
    /**
     * ReShaper disable once DuplicatingLocalDeclaration
     */
    var $: JQueryStatic;

    // Creates an Ember.NativeArray from an Array Like object. Does not modify the original object.
    Ember.A is not needed if Ember.EXTEND_PROTOTYPES is true (the default value). However, it is
    recommended that you use Ember.A when creating addons for ember or when you cannot guarantee
    that Ember.EXTEND_PROTOTYPES will be true.
    */
    function A(arr?: any[]): NativeArray;

    /**
     * An instance of Ember.Application is the starting point for every Ember application. It helps to
     * instantiate, initialize and coordinate the many objects that make up your app.
     */
    class Application extends Namespace {
        static detect(obj: any): boolean;
        static detectInstance(obj: any): boolean;
        /**
         * Iterate over each computed property for the class, passing its name and any
         * associated metadata (see metaForProperty) to the callback.
         */
        static eachComputedProperty(callback: Function, binding: {}): void;
        /**
         * Returns the original hash that was passed to meta().
         */
        @param key property name
        static metaForProperty(key: string): {};
        static isClass: boolean;
        static isMethod: boolean;
        static initializer(arguments?: ApplicationInitializerArguments): void;
{
  "name": "mylibrary",
  "define": {
    "point": {
      "x": "number",
      "y": "number"
    }
  },
  "MyConstructor": {
    "type": "fn(arg: string)",
    "staticFunction": "fn() -> bool",
    "prototype": {
      "property": "[number]",
      "clone": "fn() -> +MyConstructor",
      "getPoint": "fn(i: number) -> point"
    }
  },
  "someOtherGlobal": "string"
}