Javascript Patterns in Typescript

Javascript Idioms + Strong Typing = ???

Hi, I’m Joe

Joe DeCock

Web Developer - .NET and Front End

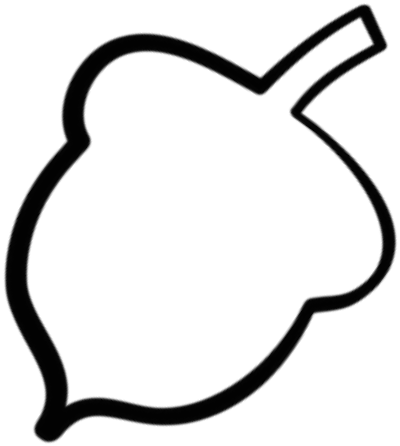
Dad

ILM Principal Consultant

@jmdc

Typescript in a Nutshell

1. Start with a highly dynamic language
2. Add static types
3. ???
4. Profit!



Topics For Today

Find out how Typescript can...

- Describe types with dynamic properties
- Type-check “magic strings” that refer to property names
- Describe functions that operate on, and return, multiple types of data

Javascript Pattern - Dynamic Properties

How can a programming language describe properties of a type without knowing the properties' names?

Examples:

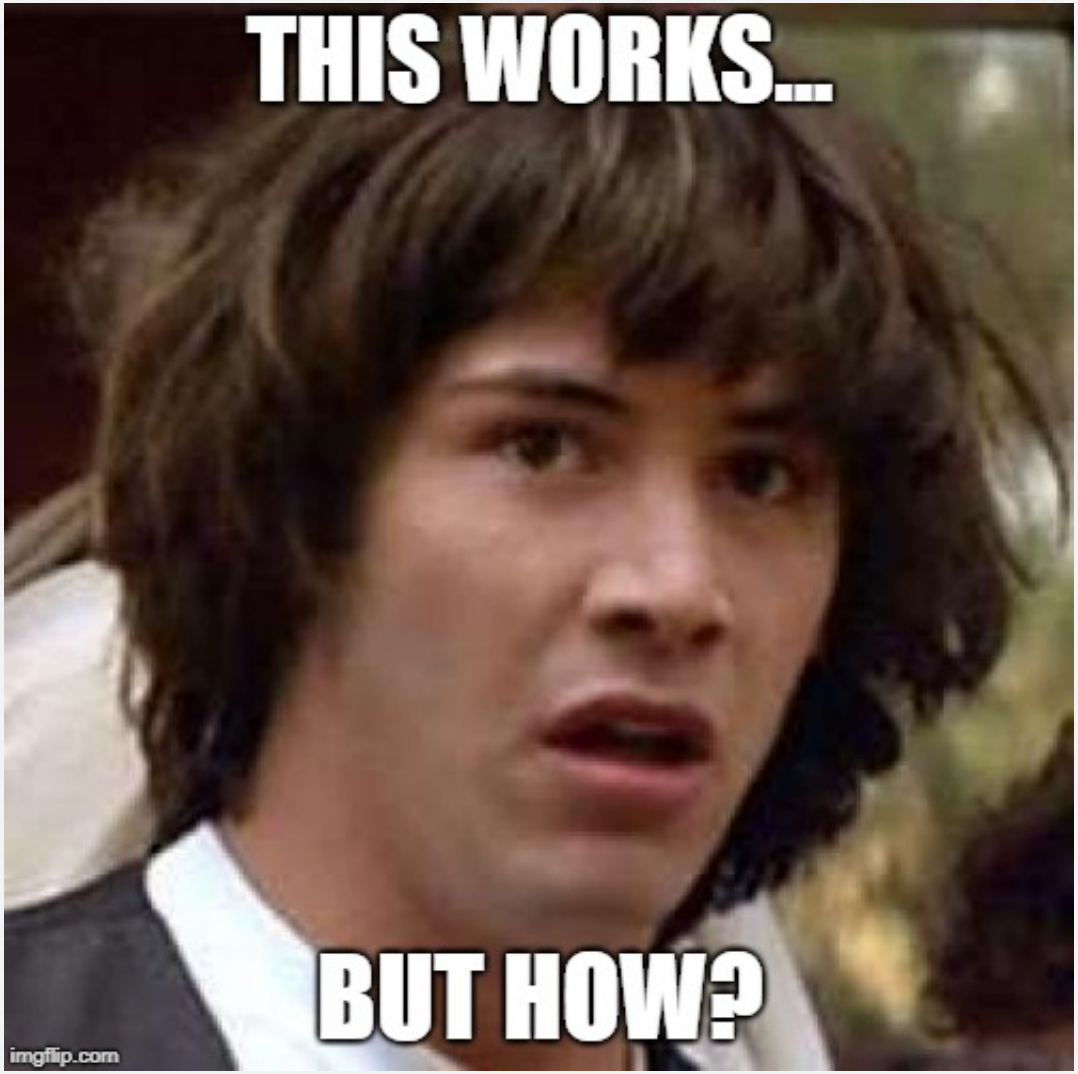
- Object literals used as dictionaries for fast lookups
- Utility libraries that create dynamic objects
- Arrays
- ArrayLike objects such as DOM NodeLists

Type safety for Lodash Dictionaries

```
const grouped = _.groupBy(todos, 'assignee');

// Somehow the type system gives the correct type to the
// Joe and Jason properties
const todosForMe: Todo[] = grouped['Joe'];
const todosForBoss: Todo[] = grouped.Jason;

// And the type system knows that this isn't allowed.
// (We need to use an array of Todos).
grouped.Amy = new Todo();
```

Type safety for array like objects

```
const nodes: NodeList = document.querySelectorAll('div');

// Type-Safe Operations
const n: Node = nodes[0];
const len: number = nodes['length'];
const n: Node = nodes.item(12);

// Not allowed by the type checker!
nodes[0] = 'asdf';
```


THIS WORKS...
BUT HOW?
imgflip.com

Index Signatures to the Rescue

Language Spec Definition:
An index signature defines a type constraint for properties in the containing type.

Syntax:

```
[key: string]: SomeType

[n: number]: SomeType
```

Typescript’s Array Declaration

```
interface Array<T> {
    length: number;
    reverse(): T[];

    // Lots of other familiar array methods

    [n: number]: T;
}
```

Lodash’s Dictionary Type

```
interface Dictionary<T> {
        [key: string]: T;
}
```

```
groupBy<T>(collection: T[], iteratee): Dictionary<T[]>;
```

Practical Use of Index Signatures

- Numeric Index Signatures
  - Rarely written
  - Help Understand syntax of index signatures - looks like array access
- String Index Signatures
  - More commonly used to describe dictionaries

Javascript Pattern:
Magic strings that refer to Property Names

How do we talk about the properties of generic types?

C#: Lambda Expressions, Reflection

Javascript: Magic Strings

Typescript: keyof operator and indexed access types

The Index Type Query Operator

Definition from the Docs: “For any type T, keyof T is the union of known, public property names of T.”

Why not just write down those property names as a type?

- Automatic updates as type T changes
- Works with generic type parameters

Operator on types

- Takes one type and produces another
- Analogous to array brackets creating array types

Example: NgRx createFeatureSelector

```
interface AppState {
        todos: TodoState;
                appointments: AppointmentState;
}


const selectTodos = createFeatureSelector<AppState, TodoState>('todos');


// We're not in Kansas anymore! (And the type checker will tell you.)
const selectTodos = createFeatureSelector<AppState, TodoState>('totos');


createFeatureSelector<T, V>(featureName: keyof T): MemoizedSelector<T, V>;
```

Indexed access operator AKA Lookup Types

Allow us to refer to the type of a property

Written like accessing elements with square brackets, except as a type

Most useful with Generic Types

Lookup Type Example: dynamic property access

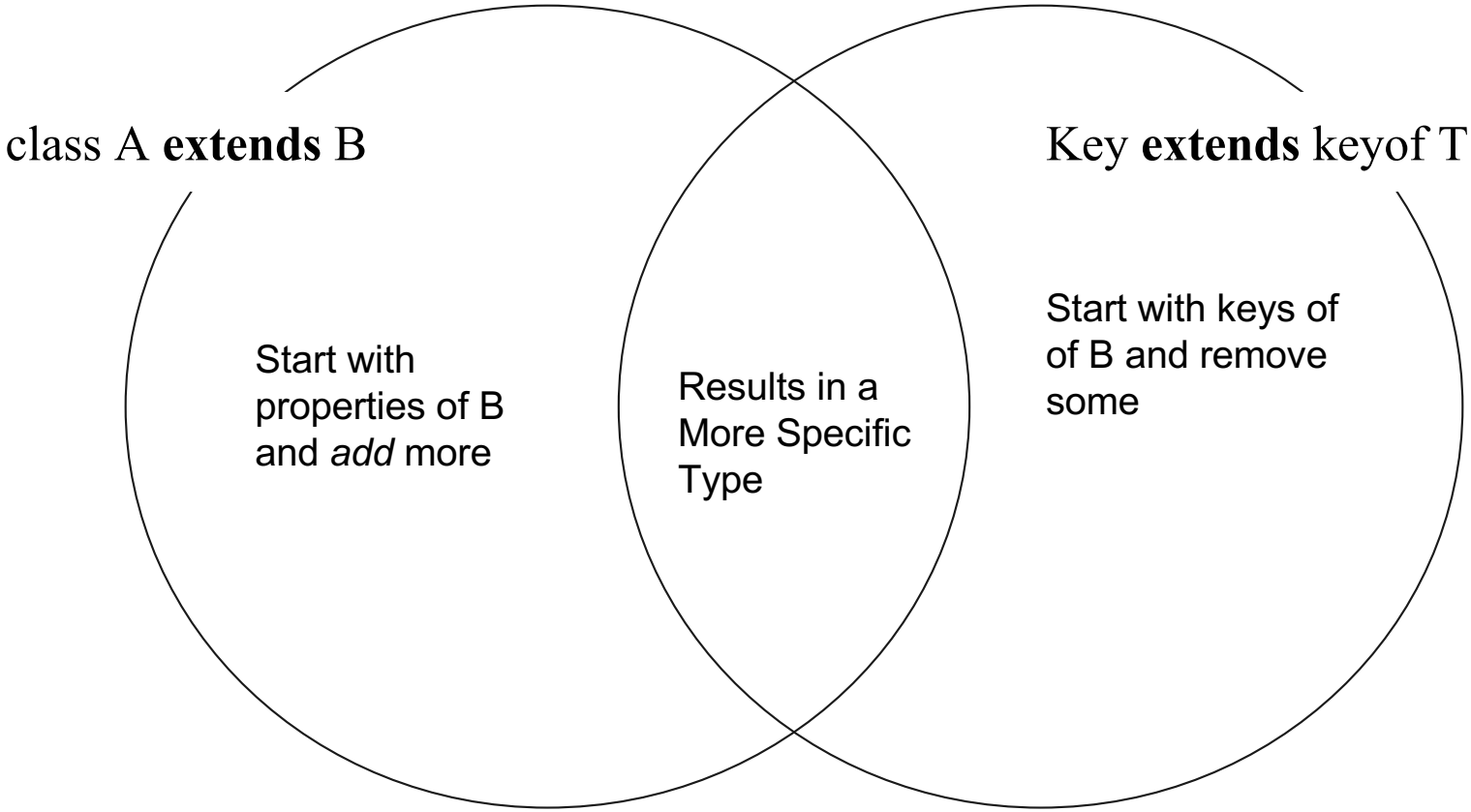
Signature:

```
get<T, Key extends keyof T>(
    object: T,
    path: Key
): T[Key];
```

Usage:

```
get(someComplexObject, 'someProperty');
```

The Extends Keyword



Javascript Pattern - Functions on multiple types

```
_.map([1,2,3]);
_.map({"a": 1, "b": 2});


moment();
moment(new Date());
moment("1985-09-29");


$("div");
$(someNode);
```

ONE FUNCTION

TO RULE THEM ALL

Union Types

Language Spec Definition:
Union types represent values that may have one of several distinct representations.

A value of a union type A | B is a value that is either of type A or type B.

Syntax:
P | Q

Example - Moment’s Constructor

```
// Simplified for legibility
moment(input: Date | string | Moment): Moment;

// Type Safe Operations
// Pass a native Date
const theFuture = moment(new Date('2015-10-15'));

// Pass a string
const myBirthday = moment('1985-09-29’);

/// Pass a moment
const backToTheFuture = moment(theFuture);
```

Example - Array Literals with mixed types of values

```
const dates = [new Date(), moment(), '1985-09-29'];
// dates' inferred type : (Date | Moment | string)[]


const myBirthday = dates[2];
// myBirthday's inferred type: Date | Moment | string
```

Okay, I've got a P | Q, how do I program with it?

- Use the common properties of P and Q
- Determine if you have a P or a Q programmatically, and then use a type assertion
- Narrow the type with type guards

Use common properties of P and Q

```
function doCoolDateLogic(myBirthday: Date | Moment | string) {

        // These are the only type safe operations:
        myBirthday.toString();
        myBirthday.valueOf();

}
```

Type Assertions

```
if((<Date>myBirthday).getFullYear) {
  ((<Date>myBirthday).getFullYear();
} else if((<string>myBirthday).substring){
  (<string>myBirthday).substring(0, 4);
} else {
  (<Moment>myBirthday).year;
}
```

Narrowing

```
if(myBirthday instanceof Date) {
  myBirthday.getFullYear();
} else if(typeof myBirthday === 'string'){
  myBirthday.substring(0, 4);
} else {
  myBirthday.year; // myBirthday is narrowed to Moment
}
```

User Defined Type Guards

```
interface MomentInputObject {
        d: number; m: number; y: number;
}

const isMomentInputObject(o: any): boolean {
        return !!o.d && !!o.m && !!o.y;
}

const isMomentInputObject(o: any): o is MomentInputObject {
        return !!o.d && !!o.m && !!o.y;
}
```

```
if(myBirthday instanceof Date) {
  myBirthday.getFullYear();
} else if(typeof myBirthday === 'string'){
  myBirthday.substring(0, 4);
} else if(isMomentInputObject(myBirthday)) {
  myBirthday.y;
} else {
  myBirthday.year;
}
```

Recap

Union Types let us talk about one type or another

Narrowing allows us to use union types in a type safe way

Three Types of Type Guards

- instanceof - classes
- typeof - primitives
- User defined type guards - interfaces

```
get<T>(url: string, o
  observe: 'response'
}): Observable<Htt
```

Overloads Make a Comeback

Javascript has no overloading

Typescript has overloads that are compiled away

To use overloads:

- Implement a function in a permissive way
- Describe how to call it with more specificity as overloads

Overload Syntax

```
// Overloads
function identity(x: string): string;
function identity(x: number): number;

// Implementation
function identity(x: string|number): string | number {
                return x;
}
```

Example - Angular’s Http Client

```
get<T>(url: string): Observable<T>;

get<T>(url: string, options: {
  observe: 'response';
}): Observable<HttpResponse<T>>;

get(url: string): Observable<Object>;
```

Recap of Overloads

Relate different ways of calling a function to different return types

Only exist in the type system

Under the covers, still *One Function to Rule Them All*

Overloads must be compatible with the implementation

Final Thought: Weird Type Errors

Extract Subexpressions and check their types

Add type annotations

Add inferred type parameters

Read library type declarations


ANY QUESTIONS?
memegenerator.net

Credits and Further Reading

Typescript Docs

https://www.typescriptlang.org/docs

Typescript Spec

https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md

Release Notes

“What’s New” section of the docs

Brian Terlson - Strongly Typed Event Emitters

https://medium.com/@bterlson/strongly-typed-event-emitters-2c2345801de8