Are you my type?

Discovering advanced types in TypeScript

A Tech Talk By:

Blake Zimmerman
Bowen Deng

What is TypeScript?

- Statically typed superset of JavaScript
- Compiles to JavaScript
- Can be used with any front-end framework
- Can be used on the back-end with Node.js

Basic Usage

```
function add(x: number, y: number): number {
  return x + y;
}

// [ts] Argument of type '"1"' is not assignable
// to parameter of type 'number'.
add("1", 1);
```

Would have returned “11” at runtime in JavaScript

“TypeScript slows me down”

```tsx
interface IProps { }
interface IState { inputValue: string; }

export class MyComponent extends React.Component<IProps, IState> {
  onInput = (event: React.SyntheticEvent<HTMLInputElement>) => {
    this.setState({
      inputValue: event.currentTarget.value
    });
  };

  render() {
    return (
      <div>
        <input onChange={this.onInput} />
      </div>
    );
  }
}
```

Purpose of this Talk

- Convince you that TypeScript is worth the learning curve
- Show the expressiveness of the language’s type system
- Empower you to write scalable code

Agenda

- Understanding the Type System
- Complex Types
- Generic Types
- Q & A

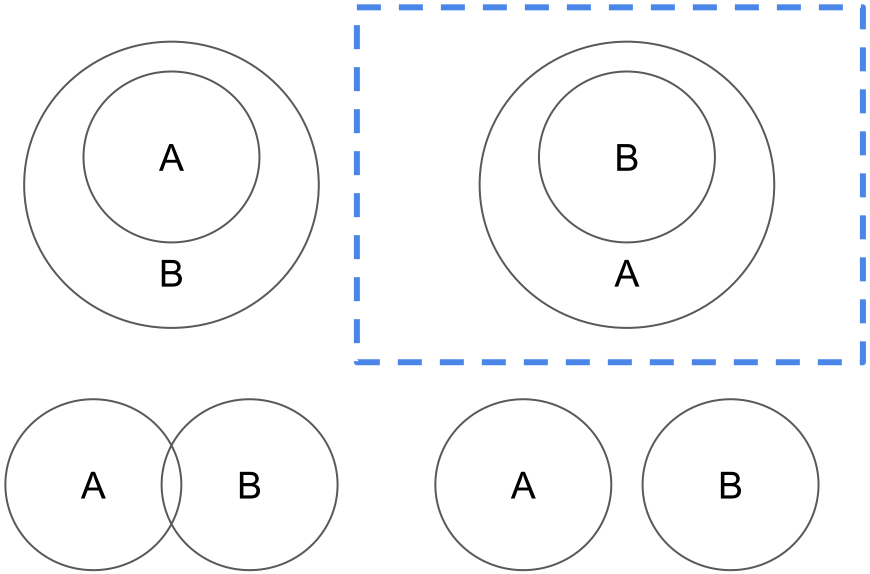
Understanding the Type System (A Set Perspective)

Think of Type as a Set

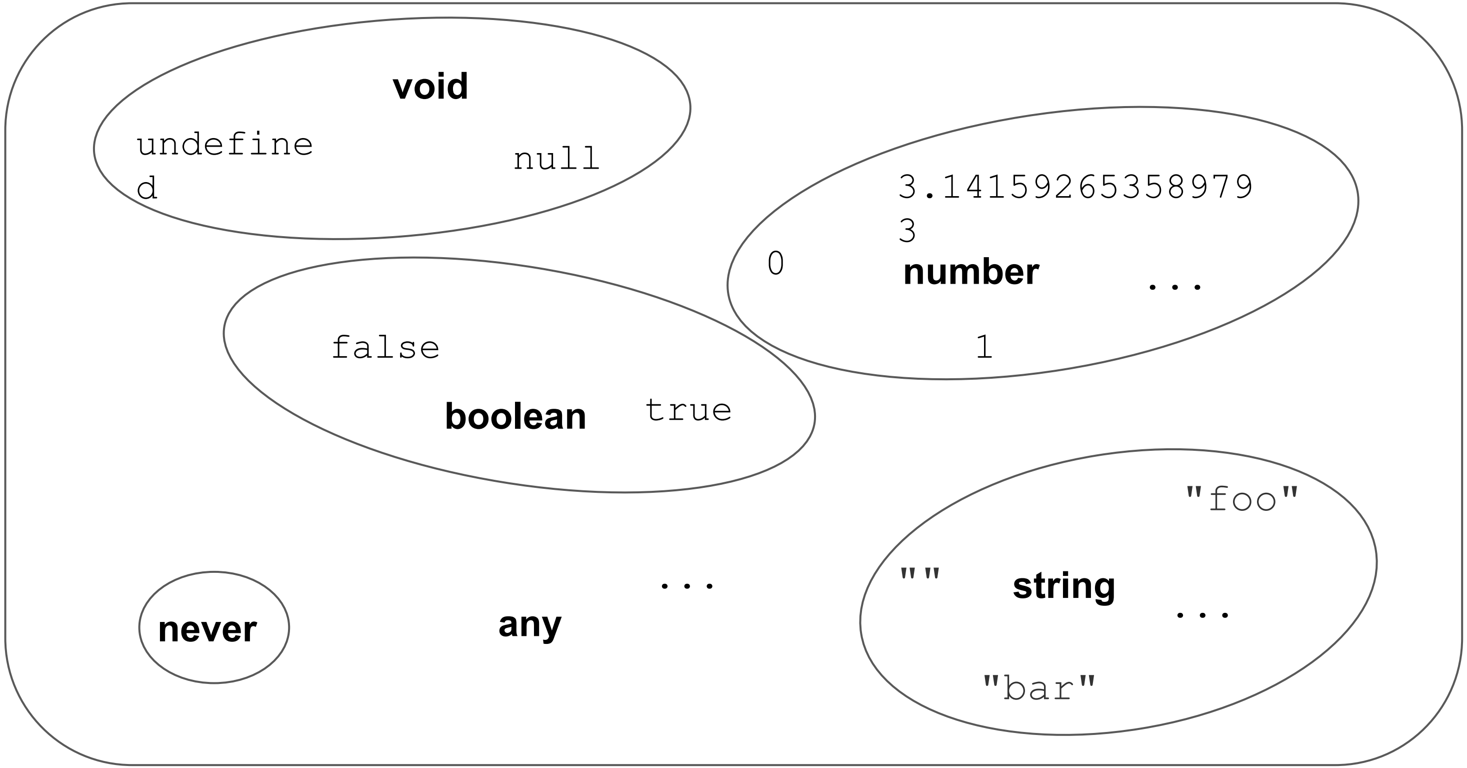
- Group of Values
- Type Declaration vs Membership
  - `const a : number = 1.0; // OK, 1.0 ∈ number`
  - `const b : number = "foo"; // Error, "foo" ∉ number`

Think of Type as a Set

- Type Compatibility vs Subsets
  - ```
    let foo: A;
    let bar: B;
    foo = bar; // OK?
    If B is a subtype of A
    ```


    ```

Value === Unit Type



Types vs Sets

Types	Sets
`extends` (Assignable / Subtype)	Subset (⊆)
`never`	Empty Set (∅)
`any`	Universe Set (U)
`|`	Union (∪)
`&`	Intersection (∩)

Describing Sets

- Extensional Definition
  - C = { 4, 2, 1, 3 }
  - D = { blue, white, red }
- Intensional Definition
  - A is the set whose members are the first four positive integers
  - B is the set of colors of the French flag

Describing Types

- Extensional Definition
  - ```
    type RGB = "red" | "green" | "blue"
    // RGB is the type which contains "red", "green" and "blue"
    ```
- Intensional Definition
  - ```
    type A = { x: number };
    // A is the type whose members are all the objects with
    // a x field whose type is number
    ```

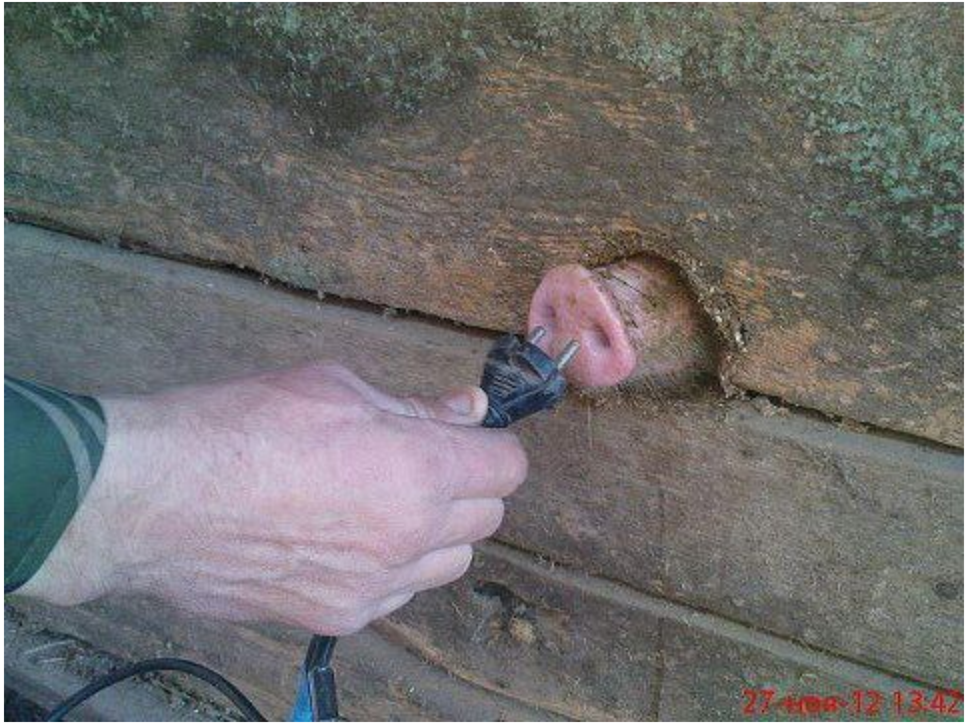
Intensional Definition => Structural Typing

- Duck typing

  "If it walks like a duck and it quacks like a duck, then it must be a duck"

- Shape Equivalence

```
type foo = { x: number };
type bar = { x: number };
```



Function Subtypes (Replaceable)

- `(t: A) => B extends (t: X) => Y`
  - ~~A extends X~~ or X extends A
  - B extends Y or ~~Y extends B~~

Complex
(...yet concrete...)
Types

Tuple Types

- Allows you to assert length of array and type of each element

```
let x: [string, number];

x = ["test", 42]; // Okay

x = [42, "test"]; // Error

x = ["test", 42, "extra"]; // Error
```

Literal Unions

- Allows you to narrow the `string` and `number` types
- Only specified literal values can be assigned to the type

```
let greeting: "hello" | "hey" | "hi";

let rating: 1 | 2 | 3 | 4 | 5;
```

String Literal Union Example

```typescript
function calculate(
  x: number,
  y: number,
  operation: "add" | "subtract"
): number {
  switch (operation) {
    case "add":
      return x + y;
    case "subtract":
      return x - y;
  }
}

calculate(4, 2, "multiply"); // Error
```

```typescript
function calculate(
  x: number,
  y: number,
  operation: "add" | "subtract"
): number {
  switch (operation) {
    case "add":
      return x + y;
    case "subtact":
      return x - y;
  }
}
```

Enums

- TypeScript has both numeric and string enums
- Similar use cases to literal unions
- Enums have runtime representations unless declared as `const enum`
- `const enum` inlines values and has no runtime representation
- Can be used as types

Numeric Enums

- First member starts at 0 by default and increments

```
enum Operations {
  Add,
  Subtract,
  Multiply,
  Divide
}

Operations.Add // returns 0
Operations.Subtract // returns 1
```

```
const enum Operations {
  Add = 1,
  Subtract,
  Multiply,
  Divide
}

Operations.Add // returns 1
Operations.Subtract // returns 2
```

String Enums

```typescript
enum Operations {
  Add = "Add",
  Subtract = "Subtract",
  Multiply = "Multiply",
  Divide = "Divide"
}

Operations.Add; // "Add" at runtime

const operation1: Operations = Operations.Add; // Okay
const operation2: Operations = "Add"; // Error

// Can be iterated over
for (const operation in Operations) {
  console.log(operation);
}
```

Interfaces

- Allows us to name type structures
- Can be extended

```
interface Point2D {
  x: number;
  y: number;
}

interface Point3D extends Point2D {
  // x: number;
  // y: number;
  z: number;
}
```

Nullable vs. Optional

- Nullable means the property is required but can be undefined
- Optional means the property might not be supplied in the object literal

```
interface A {
  a: string;
  b: number;
  c: boolean | undefined;
}
```

Nullable Property

```
interface B {
  a: string;
  b: number;
  c?: boolean;
}
```

Optional Property

Nullable vs. Optional

```typescript
interface A {
  a: string;
  b: number;
  c: boolean | undefined;
}

// Okay
const a1: A = {
  a: "test",
  b: 1,
  c: undefined
};

// Error
const a2: A = {
  a: "test",
  b: 1
};
```

vs.

```typescript
interface B {
  a: string;
  b: number;
  c?: boolean;
}

// Okay
const b1: B = {
  a: "test",
  b: 1,
  c: undefined
};

// Okay
const b2: B = {
  a: "test",
  b: 1
};
```

Discriminated Unions

- Building DUs in TypeScript requires three ingredients

1. Types that have a common, singleton type property — the *discriminant*.
2. A type alias that takes the union of those types — the *union*.
3. Type guards on the common property.

Discriminated Union Example

```
interface Square {
  type: "Square";
  size: number;
}

interface Rectangle {
  type: "Rectangle";
  width: number;
  height: number;
}

interface Circle {
  type: "Circle";
  radius: number;
}

type Shape = Square | Rectangle | Circle;
```

Pattern Matching

```
function area(s: Shape) {
  switch (s.type) {
      case "Square": return s.size * s.size;
      case "Rectangle": return s.height * s.width;
      case "Circle": return Math.PI * s.radius ** 2;
  }
}
```

(parameter) s: Square

Pattern Matching

```
function area(s: Shape) {
  switch (s.type) {
      case "Square": return s.size * s.size;
      case "Rectangle": return s.height * s.width
      case "Circle": return Math.PI * s.radius **
      default: throw new Error(`Invalid shape, ${s}`);
  }
}
```

```
(parameter) s: never
```

keyof Keyword

- Returns a string literal union of an interface’s properties

```
interface MyObject {
  field1: string;
  field2: number;
  field3: boolean;
}
        type Keys = "field1" | "field2" | "field3"
type Keys = keyof MyObject;
```

Lookup Types

- Allows you to look up the type of a property on an object

```
interface MyObject {
  field1: string;
  field2: number;
  field3: boolean;
}

type Keys = keyof MyObject;
        type Field1 = string
type Field1 = MyObject["field1"];
```

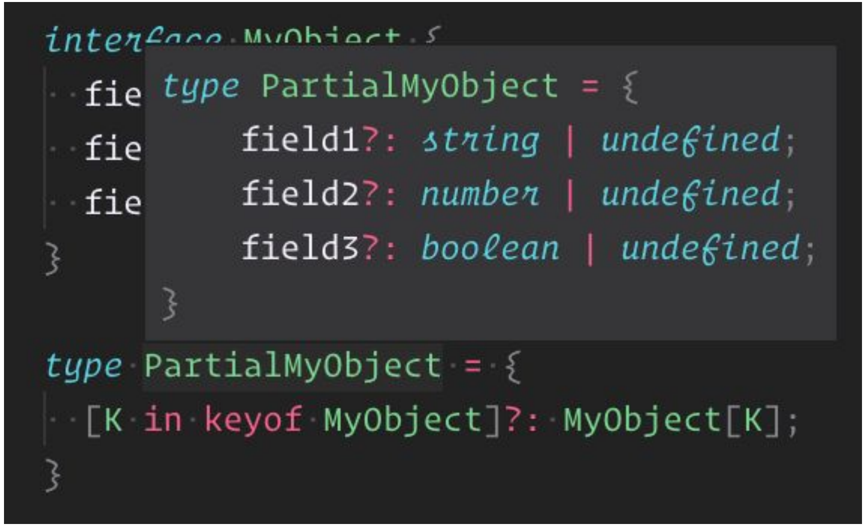
```
interface MyObject {
  field1: string;
  field2: number;
  field3: boolean;
}

type Keys = keyof MyObject;
        type Values = string | number | boolean
type Values = MyObject[keyof MyObject];
```

Mapped Types

- Allows you to map over a type to generate a new type
- Very powerful with generics

Map over an interface to make its properties → optional.



Built In Mapped Types (Generics Preview)

- TypeScript includes generic mapped types that come up a lot

Speaking of Generics....

```
type Partial<T> = {
  // Make all properties optional
  [P in keyof T]?: T[P];
}

type Readonly<T> = {
  // Make all properties readonly
  readonly [P in keyof T]: T[P];
}

type Pick<T, K extends keyof T> = {
  // Select specific properties
  [P in K]: T[P];
}

type Record<K extends string, T> = {
  // Create object with given keys and given value type
  [P in K]: T;
}
```

Generic Types

Example - Generic Map

```
interface Map<K, V> {
    clear(): void;
    delete(key: K): boolean;
    forEach(callbackfn: (value: V, key: K, map: Map<K, V>) => void, thisArg?: any): void;
    get(key: K): V | undefined;
    has(key: K): boolean;
    set(key: K, value: V): this;
    readonly size: number;
}
```

Template

Placeholders

Generic Types as Functional Programing

TypeScripts Type System is Turing Complete #14833

Open hediet opened this issue on Mar 23, 2017 · 31 comments

hediet commented on Mar 23, 2017 • edited

This is not really a bug report and I certainly don't want TypeScripts type system being restricted due to this issue. However, I noticed that the type system in its current form (version 2.2) is turing complete.

Turing completeness is being achieved by combining mapped types, recursive type definitions, accessing member types through index types and the fact that one can create types of arbitrary size. In particular, the following device enables turing completeness:

```
type MyFunc<TArg> = {
  "true": TrueExpr<MyFunction, TArg>,
  "false": FalseExpr<MyFunc, TArg>
}[Test<MyFunc, TArg>];
```

with `TrueExpr`, `FalseExpr` and `Test` being suitable types.

Even though I didn't formally prove (edit: in the meantime, I did - see below) that the mentioned device makes TypeScript turing complete, it should be obvious by looking at the following code example that tests whether a given type represents a prime number:

Parametric Polymorphism
Type Constructor
<T extends number | string>**(x: T) => T**
Type Parameter
<**T** extends number | string>(x: T) => T
Type Constraint
<T **extends number | string**>(x: T) => T

Type Constructors

Object Types	`{ x: T, y: U }`
Union Types	`T | U`
Intersection Types	`T & U`
Index Types (Query)	`keyof T`
Index Types (Indexed Access)	`T[K]`
Mapped Types	`{ [P in K]: X }`
Conditional Types	`T extends U ? X : Y`
Tuple Types	`[X, Y]`
Function Types	`(x: T) => U`
Infer Declaration	`infer U`

Conditional Types

- Non-uniform type mappings
  - `T extends U ? X : Y`
  - When T is assignable to U the type is X, otherwise the type is Y.
- Distributive
  - `A | B | C extends U ? X : Y =>`
    `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`
- Higher order type equivalences
  - `T | never ⇔ T`
  - `T & never ⇔ never`
  - `T | any ⇔ any`
  - `T & any ⇔ T`
  - `(A | B) & (C | D) ⇔ (A & C) | (A & D) | (B & C) | (B & D)`
  - `keyof (A & B) ⇔ keyof A | keyof B`
  - `keyof (A | B) ⇔ keyof A & keyof B`
  - `...`

Conditional Types - Examples

```
/**
 * Exclude from T those types that are assignable to U
 */
type Exclude<T, U> = T extends U ? never : T;

/**
 * Extract from T those types that are assignable to U
 */
type Extract<T, U> = T extends U ? T : never;

type Foo = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">;  // "b" | "d"

type Bar = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">;  // "a" | "c"

/**
 * Exclude null and undefined from T
 */
type NonNullable<T> = Exclude<T, null | undefined>;

type Id = NonNullable<number | null>;   // number
```

Open the Black Box

- Index Types
  - `keyof T`
  - `T[K]`
- Mapped Types
  - `{ [P in K]: X }`
- Infer Declaration
  - `infer U`

Reference by Key - Mapped Types & Index Types

- `{ [P in K]: X }`
  - `P`: type variable, which gets bound to each property in turn
  - `K`: string literal union, which contains the names of properties to iterate over
  - `X`: resulting type, which can reference `P`
- `keyof T`
  - index type query operator
  - returns a string literal union type of T's public property names
- `T[K]`
  - indexed access operator
  - `K extends keyof T`

```
type Partial<T> = {
    [P in keyof T]?: T[P];
};

type Required<T> = {
    [P in keyof T]-?: T[P];
};
```

Reference by Shape - Infer Declaration

- Introduce a type variable to be inferred
- Examples
  - Function
    - `type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;`
  - Type instances of generic types
    - `type Unpacked<T> = T extends Promise<infer U> ? U : T;`

Thank You!

Any Questions?

Code and Slides: https://github.com/blakezimmerman/TypeScript-Tech-Talk