01.03.2023

TypeScript 4

Mikhail Stepanov

keyof

TypeScript’s answer to the JavaScript’s Object.keys method

```ts
type Colors = {
  red: string;
  green: string;
  blue: string;
}
type Color = keyof Colors; // "red" | "green" | "blue"
```

typeof

TypeScript's answer to the JavaScript's typeof operator
(although JS typeof operator is very limited)

```ts
const color = {
  red: "#ff0000",
  green: "#00ff00",
  blue: "#0000ff",
}

// Get javascript type
console.log(typeof color); // "object"

// Extract type from a constant variable
type Colors = typeof color; // { red: string, green: string, blue: string }
```

Template Literal Types

Build on string literal types, and have the ability to expand into many strings via unions.

```ts
type MessageType = "error" | "notification" | "request";
type Lang = "en" | "fr" | "de" | "jp" | "cn";

type MessageTypeTranslations = `${MessageType}_${Lang}`;
// "error_en" | "error_fr" ... "notification_en" ...
```

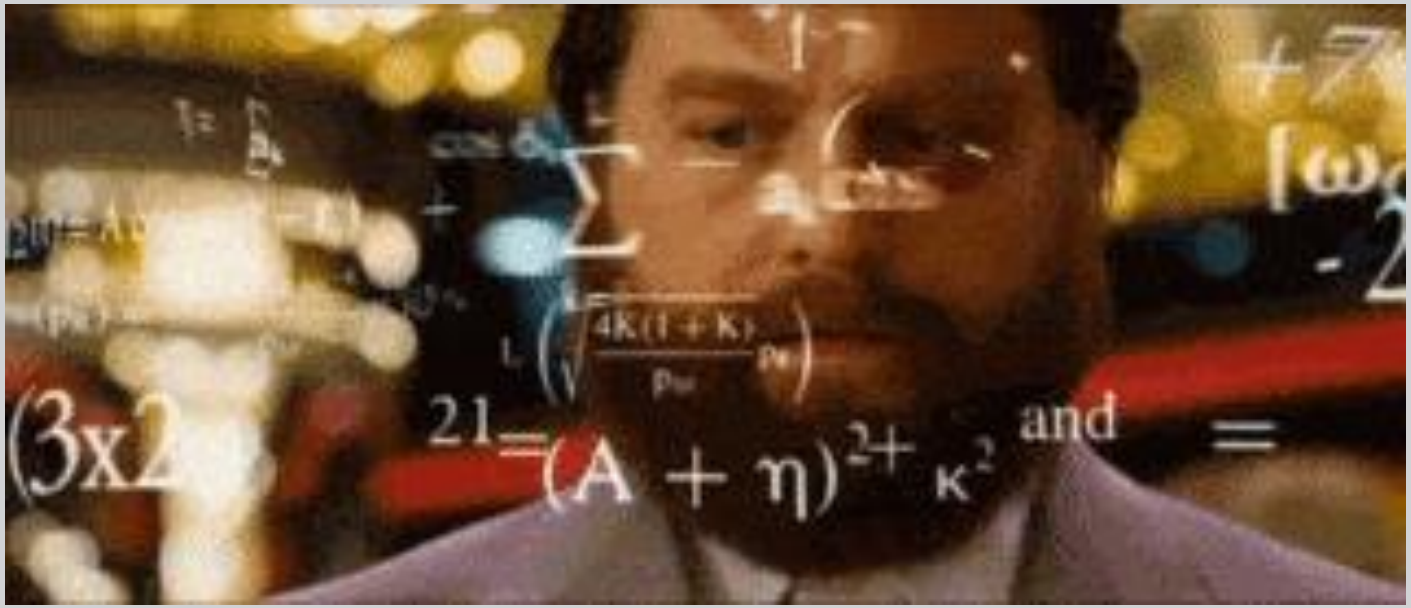
Mapped Types

A mapped type is a generic type which uses a union of PropertyKeys (frequently created via a keyof) to iterate through keys to create a type

```
type OptionsFlags<Type> = {
    [Property in keyof Type]: boolean;
};
```

There are two additional modifiers which can be applied during mapping: **readonly** and **?** which affect mutability and optionality respectively.

Type challenge



[CodeSandbox]

Discriminated unions

When every type in a union contains common property with literal types,
TS considers it a discriminated union, and can narrow out members of the union.

```ts
interface BooleanId {
    type: "boolean",
    id: boolean;
}
interface StringId {
    type: "string",
    id: string;
}

type StringOrBooleanId = BooleanId | StringId;

function getIdString(idObject: StringOrBooleanId) {
    if (idObject.type === "boolean") { /* boolean logic */ }
    if (idObject.type === "string") { /* string logic */ }
}
```

Conditional types

Conditional types are a lot like a javascript's ternary operator. Based on the condition, Typescript will decide which type can be assigned to the variable.

```
SomeType extends OtherType ? TrueType : FalseType;
```

```
// Infer id type from the usage
type ExtractIdType<T> = T extends {id: infer U} ? U : never
```

satisfies

The new **satisfies** operator lets us validate that the type of an expression matches some type, without changing the resulting type of that expression.

```ts
type Colors = "red" | "green" | "blue";
type RGB = [red: number, green: number, blue: number];
const palette = {
    red: [255, 0, 0],
    green: "#00ff00",
    bleu: [0, 0, 255]
//  ~~~~ The typo is caught!
} satisfies Record<Colors, string | RGB>;
// Both of these methods are accessible!
const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
```

Function overloading

Some JS functions can be called in a variety of argument counts and types.
In TypeScript, we can specify a function that can be called in different ways by writing overload signatures.

```typescript
// Overload signatures
function printId(id: string): string[];
function printId(id: boolean): string;

// function accepts both string and boolean
function printId(id: string | boolean): string | string[] {
    if (typeof id === "string") { /* return type string[]; */ }
    if (typeof id === "boolean") { /* return type string; */ }
}
```

Questions


Evolution
Engineering
ACCELERATING
EVOLUTION