



Advanced TypeScript Unit 3

- Conditional types
- Mapped types
- Template literal types



Conditional types



Conditional types

- If condition for types
- Can be used in generics to derive a type based on conditions
 - Union types
 - Function overloads
- Powerful with template literal & mapped types
- Possibility to infer
- Can return never



type Id<Type> = Type extends number ? {id: number} : Type



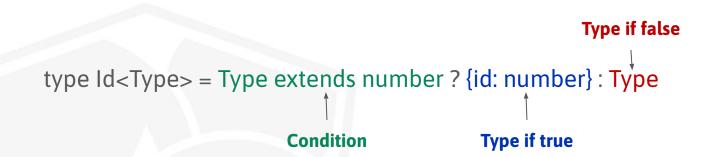
type Id<Type> = Type extends number ? {id: number} : Type

Condition











Examples

```
type Immutable<Type> = Type extends object ? Readonly<Type> : Type;

type T1 = Immutable<number>; // number

type T2 = Immutable<{ id: number; name: string }>; // {readonly id: number, readonly name: string}
```



Examples

```
type Opposite<T extends number | string> = T extends number ? string : number;

type T1 = Opposite<number>; // string

type T2 = Opposite<string>; // number
```



infer

- Only available in conditional types
- Types after extend can inferred
- Inferenced type can be returned
 - Only in the true branch
- Works on utility types, functions,...



infer against types

```
type IdType<T> = T extends { id: infer Id } ? Id : never;

type T1 = IdType<{ id: string }>; // string

type T2 = IdType<{ id: number }>; // number

type T3 = IdType<{ id: bigint }>; // bigint
```



Multiple infer against functions

```
type FnType<T> = T extends (a: infer A, b: infer B) => infer R? {args: [A, B], returned: R} : never;

const incrementer = (first: number) => first + 1

const adder = (first: number, second: number) => first + second;

type T1 = FnType<typeof incrementer> // {args: [number, undefined], returned: number}

type T2 = FnType<typeof adder> // {args: [number, number], returned: number}
```



Mapped types



Mapped types

- Generate a new type out of another
- Possibilities (in combination with conditional maps)
 - Renaming property (in combination with template literal types)
 - Changing type
 - Excluding property
 - Setting readonly
 - Setting optionally



Default mapping

```
export type Person = { id: number; firstname: string; lastname: string; birthdate: Date; };

type Clone<Type> = { [Property in keyof Type]: Type[Property] };

type ClonedPerson = Clone<Person>; // Person
```



Changing property types

```
export type Person = { id: number; firstname: string; lastname: string; birthdate: Date; };

type Getter<Type> = { [Property in keyof Type]: () => Type[Property] };

type PersonGetter<Type> = Getter<Person>;
```



Excluding properties (via never)



ReadOnly<T>, Optional<T>

```
type Partial<Type> = { [Property in keyof Type]?: Type[Property] };
type UnPartial<Type> = { [Property in keyof Type]-?: Type[Property] };

type Readonly<Type> = { readonly [Property in keyof Type]: Type[Property] };
type NotReadonly<Type> = {
    -readonly [Property in keyof Type]: Type[Property];
};
```



Template literal types



Template Literal Types

- Possibility to construct instructions via "string interpolation"
- Most advanced technique in terms of meta-programming
- Feels a little bit like eval()
- Real power in combination with mapped & conditional types



Minimalistic example

```
type LocaleMaker<COUNTRY extends string, LANG extends string> = `${COUNTRY}-${LANG}`;
type AT = LocaleMaker<"at", "de">; // "at-de"
```



Extending with unions

```
type Engine = "diesel" | "petrol" | "electric";

type Shift = "manual" | "automatical";

type CarType = "sedan" | "suv" | "coupe";

type CarConfig = `${Engine}_${Shift}_${CarType}`;
```



combi with conditional types

```
type ValidLocale<T> = T extends `${infer Country}_${infer Language}`
    ? T
    : never;

type Austria = ValidLocale<"at_de">; // at_de

type InvalidLocale = ValidLocale<"at-de">; // never
```



String utility types

```
type LocaleMaker<COUNTRY extends string, LANG extends string> = `${COUNTRY}-${LANG}`;
type AT = LocaleMaker<"at", "de">; // "at-de"
```



Combination mapped types

```
export type Person = { id: number; firstname: string; lastname: string; birthdate: Date; };

type Getter<Type> = {
   [Property in keyof Type as `get${Capitalize<Property & string>}`]: () => Type[Property];
};

type PersonGetter<Type> = Getter<Person>;
```



Showing off 1: date types to getter methods

```
export type Person = {
 id: number;
 firstname: string;
  lastname: string;
  birthdate: Date;
};
type DateGetter<Type> = {
  [Property in keyof Type as Type[Property] extends Date
    ? `get${Capitalize<Property & string>}`
    : Property]: Type[Property] extends Date
    ? () => Type[Property]
    : Type[Property];
};
// {id: number, firstname: string, lastname: string, birthdate: () => Date}
type BetterPerson<Type> = DateGetter<Person>;
```



Showing off 2: recursive typing





Case Study ngrx: createAction



Case Study ngrx: createFeature



Further Reading

- https://www.typescriptlang.org/docs/
- https://effectivetypescript.com/2020/11/05/template-literal-types/
- https://medium.com/@bytefer
- https://fettblog.eu/
- https://github.com/type-challenges/type-challenges
- https://github.com/millsp/ts-toolbelt
- https://github.com/ghoullier/awesome-template-literal-types
- https://dev.to/phenomnominal/i-need-to-learn-about-typescript-template-lite ral-types-51po
- https://www.youtube.com/watch?v=RH49aarW6sU



People to Follow

- Anders Hejlsberg
- Daniel Rosenwasser
- Stefan Baumgartner
- Matt Pocock



Recommended Libraries

- GitHub sindresorhus/type-fest: A collection of essential TypeScript types
- GitHub colinhacks/zod: TypeScript-first schema validation with static type
 inference

