Cookbook

Type Definition Generator

see here: Type Definition Generator

Use the values of an object as key type for another

one

```
import ts from 'typescript';
console.log(ts.version);
export interface SomeIDsType {
  empty: '';
  formA: '100';
  formB: '120';
  formC?: '130';
}

export enum SomeSubVariants {
  'EE' = 'e',
  'OO' = 'o',
  'PP' = 'p',
}

/**
 * What we want to achieve is, to
 * create a type for the following object,
 * from existing types or values:

    export const ResultObj = {
      '100': {
        name: 'George Michael',
        subTypes: {
          e: 'yes',
          o: 'no',
          p: 'yes',
        },
      },
      '120': {
        name: 'Michael Jackson',
        subTypes: {
          e: 'yes',
          o: 'no',
          p: 'yes',
        },
      },
    };
 *
 * "130" should be optional.
```

```
 */

type YesOrNo = 'yes' | 'no';
type SubTypesType = {
  [key in SomeSubVariants]: YesOrNo;
};

type ResultObjValueType = {
  name: string;
  subTypes: SubTypesType;
};

/**
 * Get the values of the first object
 * as keys of the second object by ignoring
 * the empty string key.
 */

type ValuesOfInterfaceAsKeyType<T> = T[keyof T] extends infer V
  ? V extends string
    ? V extends ''
      ? never
      : V
    : never
  : never;

type AllowedKeys = ValuesOfInterfaceAsKeyType<SomeIDsType>;

/**
 * These are:
 * type AllowedKeys = "100" | "120" | "130"
 */

/**
   * The following approach would not allow us
   * to make the "130" key optional.

  export type ResultObjType = {
    [key in AllowedKeys]: ResultObjValueType;
  };

  * Therfore we need to use the following approach:
```

```
    export type ResultObjType = {
    [key in keyof SomeIDsType as SomeIDsType[key] extends AllowedKeys
      ? SomeIDsType[key]
      : never]: ResultObjValueType;
  };

  * The question mark makes the key optional
  * in the example above.
  *
  * But the solution, which I like the most, does
  * not make "130" not optional neither :/
  *

  export type ResultObjType = Record<
   AllowedKeys, ResultObjValueType
  >;

  * Therefore, we need to use the following
  * complex approach:
  */

export type ResultObjType = {
  [key in keyof SomeIDsType as SomeIDsType[key] extends AllowedKeys
    ? SomeIDsType[key]
    : never]?: ResultObjValueType;
};

export const ResultObj: ResultObjType = {
  '100': {
    name: 'George Michael',
    subTypes: {
      e: 'yes',
      o: 'no',
      p: 'yes',
    },
  },
  '120': {
    name: 'Michael Jackson',
    subTypes: {
      e: 'yes',
      o: 'no',
      p: 'yes',
    },
```

```
  },
};

console.log('Result Obj:', ResultObj);
```

Conversion between Objects

```ts
export type KeyValueObjType = {
  key: string;
  value: string;
};

export type TranslationsType = {
  [index: string]: Array<KeyValueObjType>;
};

export const translations: TranslationsType = {
  de: [
    {
      key: 'SomeKey',
      value: 'Translation for Key',
    },
  ],
  en: [
    {
      key: 'SomeKey',
      value: 'Translation for Key',
    },
  ],
};

export const langKeys = Object.keys(translations);

export type LangKeysType = (typeof langKeys)[number]; // <- TRICK

/**
 * same as:
 *   export type LangKeysType = 'de' | 'en';
 *
 * But if you use it so, you have to initialise
 * these props in resultObj
 */

export type FlatValueType = {
  [key: string]: string;
};
```

```ts
export type FlatTranslationStringsType = {
  [lang in LangKeysType]: FlatValueType;
};

export const resultObj: FlatTranslationStringsType = {};

langKeys.forEach((langKey: string) => {
  const langArr: Array<KeyValueObjType> = translations[langKey];
  resultObj[langKey] = {};
  langArr.forEach((obj) => {
    resultObj[langKey][obj.key] = obj.value;
  });
});

console.log(resultObj);

/**
 * Output:
 * {
 *  de: { SomeKey: 'Translation for Key' },
 *  en: { SomeKey: 'Translation for Key' }
 * }
 */
```

Sophisticated Type Guards

```ts
type MyCustomType = `A8${string}`;

function isMyCustomType(value: string): value is MyCustomType {
  return /^A8\d{2}$/.test(value);
}

function createUniqueArray<T extends ReadonlyArray<MyCustomType>>(
  arr: T,
): T {
  const uniqueSet = new Set(arr);
  if (uniqueSet.size !== arr.length) {
    throw new Error('Duplicate entries found in array.');
  }
  return arr;
}

/**
 * If you type anything else than MyCustomType in the array,
 * you will get an error.
 */

const x = createUniqueArray(['A807', 'A811', 'A800'] as const);
```

Only 2 Letters in a special combination

```ts
type Only2Letters = `${'A' | 'B'}${string}`;

function isOnly2Letters(value: string): value is Only2Letters {
  return /^(A|B)[A-Z]{1}$/.test(value);
}

const y: Array<Only2Letters> = ['AA', 'AB', 'BA', 'BB', 'CC']; // <- Error 'CC'
```

Type Guard + RunTime Check

```ts
export type BType = `${string}${number}${number}${number}`;

export function isBType(value: string): value is BType {
  return /^A8[012]{1}[0-9]{1}$/g.test(value);
}

export const BTypeArr: Array<BType> = [
  'A800',
  'A803',
  'A813',
  'A814',
  'A823',
  'A833',
];

BTypeArr.forEach((bType) => {
  if (!isBType(bType)) {
    console.warn(`Invalid BType: ${bType}`);
    console.log(`Expected: ${/^A8[012]{1}[0-9]{1}$/g}`);
    console.log(
      `Expected: A8 followed by 0, 1 or 2 followed by any number`,
    );
  }
});
```