نمونه ای از جنریک و ترکیب آنها در typealiasing :

type OrNull<Type> = Type | null;

type OneOrMany<Type> = Type | Type[];

type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;

type OneOrManyOrNull<Type> = OneOrMany<Type> | null

type OneOrManyOrNullStrings = OneOrManyOrNull<string>;

نمونه دیگری از کاربرد جنریک در آرایه ها:

interface Array<Type> {

length: number;

pop(): Type | undefined; Removes the last element from an array and returns it

  push(...items: Type[]): number; Appends new elements to an array, and returns the new length of the array

}

داده از نوع Tuple : تعداد و نوع اجزا از قبل مشخص است و برای کاربردهایی مانند فراخوانی API هایی که لیست داده های برگشتی زیاد است کاربرد دارد.

type StringNumberPair = [string, number];

کاربرد Tuple ها در پارامترهای از نوع rest :

type StringNumberBooleans = [string, number, ...boolean[]];

type StringBooleansNumber = [string, ...boolean[], number];

type BooleansStringNumber = [...boolean[], string, number];

مثال از فراخوانی:

const a: StringNumberBooleans = ["hello", 1];

const b: StringNumberBooleans = ["beautiful", 2, true];

const c: StringNumberBooleans = ["world", 3, true, false, true, true];

یکی از کاربردهای اینگونه Tuple ها در تعریف توابع و فراخوانی آنهاست زمانی که تعداد پارامترها زیاد است. مثال:

function readButtonInput(...args: [string, number, ...boolean[]]) {

const [name, version, ...input] = args;

// ...

}

معادل حالت زیر است:

function readButtonInput(name: string, version: number, ...input: boolean[]) {

// ...

}

نحوه معرفی آرایه ای از Type ها به دو روش مختلف:

function loggingIdentity<Type>(arg: Type[]): Type[] {

console.log(arg.length);

return arg;

}

function loggingIdentity<Type>(arg: Array<Type>): Array<Type> {

console.log(arg.length); // Array has a .length, so no more error

return arg;

}

میتوان نحوه رفتار یک تابع (ورودی ها و خروجی آن) را تحت عنوان یک Interface معرفی کرد. مثال:

interface GenericIdentityFn<Type> {

(arg: Type): Type;

}

function identity<Type>(arg: Type): Type {

return arg;

}

let myIdentity: GenericIdentityFn<number> = identity;

محدود کردن دسترسی در کلاسهای جنریک به بخش خاصی از Object ها:

function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {

return obj[key];

}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a"); ok

getProperty(x, "m"); Error

بعضی از کاربردهای Typeof و keyof :

type Point = { x: number; y: number };

type P = keyof Point; => is the same as type P = “x” | “y”;

اما وقتی این عملگر را در کنار index signature ها به کار میبریم به نوع اشاره دارد. مثال :

type Arrayish = { [n: number]: unknown };

type A = keyof Arrayish; => is the same as type A = number;

برای type of به صورت زیر خواهد بود:

let s = "hello";

let n: typeof s; => is the same as let n: string

عملگر ReturnType به مقدار برگشتی یک Type اشاره میکند. به عنوان مثال:

type Predicate = (x: unknown) => boolean;

type K = ReturnType<Predicate>; => is the same as type K=Boolean

این عملگر برای چک کردن مقدار خروجی توابع به صورت زیر به کار می رود:

function f() {

return { x: 10, y: 3 };

}

type P = ReturnType<typeof f>; => type P = {

x: number;

y: number;

}

Conditional Types : انواع داده های شرطی که بر اساس شرایط تفاوت میکنند. وقتی این قابلیت با جنریک ترکیب میشود کاربردی تر می شود. به عنوان مثال دو تکه کد زیر با هم معادل می باشند.

|  |
| --- |
| interface IdLabel {  id: number /\* some fields \*/;  }  interface NameLabel {  name: string /\* other fields \*/;  }    function createLabel(id: number): IdLabel;  function createLabel(name: string): NameLabel;  function createLabel(nameOrId: string | number): IdLabel | NameLabel;  function createLabel(nameOrId: string | number): IdLabel | NameLabel {  throw "unimplemented";  } |
| interface IdLabel {  id: number /\* some fields \*/;  }  interface NameLabel {  name: string /\* other fields \*/;  }  type NameOrId<T extends number | string> = T extends number  ? IdLabel  : NameLabel;  function createLabel<T extends number | string>(NameOrId: T): NameOrId<T> {  throw "unimplemented";  } |

Mapped Types : زمانی که قرار نیست خودمان را تکرار کنیم و نوع داده های جدید از روی نوع داده های قبلی بسازیم، استفاده می­شود و از Index Signature در طول انجام آن استفاده می­شود. مثال:

type OptionsFlags<Type> = {

[Property in keyof Type]: boolean;

};

type Features = {

darkMode: () => void;

newUserProfile: () => void;

};

type FeatureOptions = OptionsFlags<Features>; => type FeatureOptions = {

darkMode: boolean;

newUserProfile: boolean;

}

هنگام Mapping میتوان از modifier های optional و mutable استفاده کرد. نکته اینکه اگر بخواهیم یکی از ویژگیهای فوق را از یک type بگیریم باید علامت – و اگر بخواهیم چیزی را به آن اضافه کنیم علامت + را به کار میبریم(اگر علامتی گذاشته نشود به معنای + خواهد بود). مثال:

type CreateMutable<Type> = {

-readonly [Property in keyof Type]: Type[Property];

};

type LockedAccount = {

readonly id: string;

readonly name: string;

};

type UnlockedAccount = CreateMutable<LockedAccount>;

* type UnlockedAccount = { id: string;

name: string;}

مثال دیگری در مورد حذف Optional به شرح زیر می باشد:

type Concrete<Type> = {

[Property in keyof Type]-?: Type[Property];

};

type MaybeUser = {

id: string;

name?: string;

age?: number;

};

type User = Concrete<MaybeUser>;

* type User = {

id: string;

name: string;

age: number;

}

یکی از کاربردهای دیگر mapping به این صورت است که با استفاده از عملگر as می توان نام و مشخصات دیگر type را عوض کرد. مثلا در تکه کد زیر به نام تمامی فیلدهای کلاس Person یک get اضافه کرده، حرف اول آن را Capitalize کرده و آنها را به توابعی بدون پارامتر با خروجی برابر با Type آن فیلد تبدیل میکند.

type Getters<Type> = {

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

};

interface Person {

name: string;

age: number;

location: string;

}

type LazyPerson = Getters<Person>;

* type LazyPerson = {

getName: () => string;

getAge: () => number;

getLocation: () => string;

}

یا مثلا میتوان یک تا تعدادی از فیلدهایی که قرار است Mapping بر اساس آنها انجام شود را با استفاده از فیلترهای مختلف حذف کرد. مثال:

type RemoveKindField<Type> = {

[Property in keyof Type as Exclude<Property, "kind">]: Type[Property]

};

interface Circle {

kind: "circle";

radius: number;

}

type KindlessCircle = RemoveKindField<Circle>;

* type KindlessCircle = {

radius: number;

}

یا مثال دیگری بر همین اساس که بر اساس یک شرط فیلترینگ را انجام داده و نوع فیلد را تغییر میدهد، در زیر آمده است:

type ExtractPII<Type> = {

[Property in keyof Type]: Type[Property] extends { pii: true } ? true : false;

};

type DBFields = {

id: { format: "incrementing" };

name: { type: string; pii: true };

};

type ObjectsNeedingGDPRDeletion = ExtractPII<DBFields>;

* type ObjectsNeedingGDPRDeletion = {

id: false;

name: true;

}