## *Using type predicates:*

To define a user-defined type guard, we simply need to define a function whose return type is a type predicate.

function isFish(pet: Fish | Bird): pet is Fish {

return (pet as Fish).swim !== undefined;

}

const zoo: (Fish | Bird)[] = [getSmallPet(), getSmallPet(), getSmallPet()];

const underWater1: Fish[] = zoo.filter(isFish);

// or, equivalently

const underWater2: Fish[] = zoo.filter(isFish) as Fish[];

// The predicate may need repeating for more complex examples

const underWater3: Fish[] = zoo.filter((pet): pet is Fish => {

if (pet.name === "sharkey") return false;

return isFish(pet);

});

*Never Type:*

The never type is assignable to every type; however, no type is assignable to never (except never itself). This means you can use narrowing and rely on never turning up to do exhaustive checking in a switch statement.

*Generic Functions:*

In TypeScript, generics are used when we want to describe a correspondence between two values. We do this by declaring a type parameter in the function signature.

It’s common to write a function where the types of the input relate to the type of the output, or where the types of two inputs are related in some way.

function firstElement<Type>(arr: Type[]): Type | undefined {

return arr[0];

}

function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {

return arr.map(func);

}

// Parameter 'n' is of type 'string'

// 'parsed' is of type 'number[]'

const parsed = map(["1", "2", "3"], (n) => parseInt(n));

Constraints

We’ve written some generic functions that can work on any kind of value. Sometimes we want to relate two values, but can only operate on a certain subset of values. In this case, we can use a constraint to limit the kinds of types that a type parameter can accept.

function longest<Type extends { length: number }>(a: Type, b: Type) {

if (a.length >= b.length) {

return a;

} else {

return b;

}

}

// longerArray is of type 'number[]'

const longerArray = longest([1, 2], [1, 2, 3]);

// longerString is of type 'alice' | 'bob'

const longerString = longest("alice", "bob");

// Error! Numbers don't have a 'length' property

const notOK = longest(10, 100);

Remember, type parameters are for relating the types of multiple values. If a type parameter is only used once in the function signature, it’s not relating anything. This includes the inferred return type

Using the readonly modifier doesn’t necessarily imply that a value is totally immutable - or in other words, that its internal contents can’t be changed. It just means the property itself can’t be re-written to.

interface Home {

readonly resident: { name: string; age: number };

}

*Index Signatures*

Sometimes you don’t know all the names of a type’s properties ahead of time, but you do know the shape of the values.

interface StringArray {

[index: number]: string;

}

interface ReadonlyStringArray {

readonly [index: number]: string;

}

*Generic Object Types:*

interface Box<Type> {

contents: Type;

}

Box is reusable in that Type can be substituted with anything. That means that when we need a box for a new type, we don’t need to declare a new Box type at all.

interface Box<Type> {

contents: Type;

}

interface Apple {

// ....

}

// Same as '{ contents: Apple }'.

type AppleBox = Box<Apple>;

function setContents<Type>(box: Box<Type>, newContents: Type) {

box.contents = newContents;

}

It is worth noting that type aliases can also be generic.

type Box<Type> = {

contents: Type;

};

Since type aliases, unlike interfaces, can describe more than just object types, we can also use them to write other kinds of generic helper types.

type OrNull<Type> = Type | null;

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

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

*The Array Type:*

Much like the Box type above, Array itself is a generic type.

interface Array<Type> {

length: number;

pop(): Type | undefined;

push(...items: Type[]): number;

// ...

}

Modern JavaScript also provides other data structures which are generic, like Map<K, V>, Set<T>, and Promise<T>. All this really means is that because of how Map, Set, and Promise behave, they can work with any sets of types.

*The ReadonlyArray Type:*

The ReadonlyArray is a special type that describes arrays that shouldn’t be changed.

Just as TypeScript provides a shorthand syntax for Array<Type> with Type[], it also provides a shorthand syntax for ReadonlyArray<Type> with readonly Type[].

*Tuple Types:*

A tuple type is another sort of Array type that knows exactly how many elements it contains, and exactly which types it contains at specific positions.

type StringNumberPair = [string, number];

type Either2dOr3d = [number, number, number?];

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

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

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

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

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

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

type readOnlyTuple = readonly [string, number]

*Creating Types from Types:*

TypeScript’s type system is very powerful because it allows expressing types in terms of other types. The simplest form of this idea is generics. Additionally, we have a wide variety of type operators available to use. It’s also possible to express types in terms of values that we already have. By combining various type operators, we can express complex operations and values in a succinct, maintainable way.

* *Generics* - Types which take parameters
* *Keyof Type Operator* - Using the keyof operator to create new types
* *Typeof Type Operator* - Using the typeof operator to create new types
* *Indexed Access Types* - Using Type['a'] syntax to access a subset of a type
* *Conditional Types* - Types which act like if statements in the type system
* *Mapped Types* - Creating types by mapping each property in an existing type
* *Template Literal Types* - Mapped types which change properties via template literal strings.

*Generics:*

interface GenericIdentityFn<Type> {

(arg: Type): Type;

}

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

return arg;

}

let myIdentity: GenericIdentityFn<number> = identity;

A class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class’s type parameter.

interface Lengthwise {

length: number;

}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {

console.log(arg.length); // Now we know it has a .length property, so no more error

return arg;

}

*Using Type Parameters in Generic Constraints:*

You can declare a type parameter that is constrained by another type parameter. For example, here we’d like to get a property from an object given its name. We’d like to ensure that we’re not accidentally grabbing a property that does not exist on the obj, so we’ll place a constraint between the two types:

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

return obj[key];

}

Using Class Types in Generics:

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

function create<Type>(c: { new (): Type }): Type {

return new c();

}

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

function createInstance<A extends Animal>(c: new () => A): A {

return new c();

}

*The keyof type operator:*

The keyof operator takes an object type and produces a string or numeric literal union of its keys. The following type P is the same type as type P = "x" | "y"

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

type P = keyof Point;

If the type has a string or number index signature, keyof will return those types instead

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

type A = keyof Arrayish;

type Mapish = { [k: string]: boolean };

type M = keyof Mapish;

Note that in this example, M is string | number — this is because JavaScript object keys are always coerced to a string, so obj[0] is always the same as obj["0"].

*The typeof type operator:*

JavaScript already has a typeof operator you can use in an expression context.

TypeScript adds a typeof operator you can use in a type context to refer to the type of a variable or property:

let s = "hello";

let n: typeof s; //n == string

function f() {

return { x: 10, y: 3 };

}

type P = ReturnType<typeof f>; //p=={x: number; y: number}

*Indexed access type:*

We can use an indexed access type to look up a specific property on another type:

type Person ={ age: number; name: string; alive: boolean };

type Age = Person["age"];

The indexing type is itself a type, so we can use unions, keyof, or other types entirely:

type I1 = Person["age" | "name"]; // string | number

type I2 = Person[keyof Person];// string | number | boolean

type AliveOrName = "alive" | "name";

type I3 = Person[AliveOrName]; // string | Boolean

Another example of indexing with an arbitrary type is using number to get the type of an array’s elements. We can combine this with typeof to conveniently capture the element type of an array literal:

const MyArray = [

{ name: "Alice", age: 15 },

{ name: "Bob", age: 23 },

{ name: "Eve", age: 38 },

];

type Person = typeof MyArray[number];//{name:string; age:number}

type Age = typeof MyArray[number]["age"]; // number

type Age2 = Person["age"];

Conditional Types:

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>(idOrName: T): NameOrId<T> {

throw "unimplemented";

}

let a = createLabel("typescript"); // NameLabel

let b = createLabel(2.8); // IdLabel

let c = createLabel(Math.random() ? "hello" : 42); // NameLabel | IdLabel

Conditional Type Constraints:

type MessageOf<T> = T["message"]; // Error

In this example, TypeScript errors because T isn’t known to have a property called message. We could constrain T, and TypeScript would no longer complain

type MessageOf<T extends { message: unknown }> = T["message"];

interface Email {

message: string;

}

type EmailMessageContents = MessageOf<Email>; // string

However, what if we wanted MessageOf to take any type, and default to something like never if a message property isn’t available? We can do this by moving the constraint out and introducing a conditional type:

type MessageOf<T> = T extends { message: unknown } ? T["message"] : never;

interface Email {

message: string;

}

interface Dog {

bark(): void;

}

type EmailMessageContents = MessageOf<Email>; //string

type DogMessageContents = MessageOf<Dog>; //never

As another example, we could also write a type called Flatten that flattens array types to their element types, but leaves them alone otherwise:

type Flatten<T> = T extends any[] ? T[number] : T;

// Extracts out the element type.

type Str = Flatten<string[]>; // string

// Leaves the type alone.

type Num = Flatten<number>; // number

*Infer keyword:*

The infer keyword and conditional typing in TypeScript allows us to take a type and isolate any piece of it for later use.

The infer keyword compliments conditional types and cannot be used outside an extends clause. Infer allows us to define a variable within our constraint to be referenced or returned.

type GetFirstArgumentOfAnyFunction<T> = T extends (

first: infer FirstArgument,

...args: any[]

) => any

? FirstArgument

: never

type t = GetFirstArgumentOfAnyFunction<(name: string, age: number) => void> // string

type PromiseReturnType<T> = T extends Promise<infer Return> ? Return : T

type t = PromiseReturnType<Promise<string>> // string

type ArrayType<T> = T extends (infer Item)[] ? Item : T

type t = ArrayType<[string, number]> // string | number

*Distributive Conditional Types:*

When conditional types act on a generic type, they become distributive when given a union type. For example, take the following:

type ToArray<Type> = Type extends any ? Type[] : never;

If we plug a union type into ToArray, then the conditional type will be applied to each member of that union.

type ToArray<Type> = Type extends any ? Type[] : never;

type StrArrOrNumArr = ToArray<string | number>; // string[] | number[]

Typically, distributivity is the desired behavior. To avoid that behavior, you can surround each side of the extends keyword with square brackets.

type ToArrayNonDist<Type> = [Type] extends [any] ? Type[] : never;

// 'StrArrOrNumArr' is no longer a union.

type StrArrOrNumArr = ToArrayNonDist<string | number>; // (string | number)[]

*Mapped Types:*

When you don’t want to repeat yourself, sometimes a type needs to be based on another type.

type OptionsFlags<Type> = {

[Property in keyof Type]: boolean;

};

type Features = {

darkMode: () => void;

newUserProfile: () => void;

};

type FeatureOptions = OptionsFlags<Features>; // { darkMode:Boolean; newUserProfile: Boolean}

*Mapping Modifiers:*

There are two additional modifiers which can be applied during mapping: readonly and ? which affect mutability and optionality respectively. You can remove or add these modifiers by prefixing with - or +. If you don’t add a prefix, then + is assumed.

type CreateMutable<Type> = {

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

};

type Concrete<Type> = {

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

};

*Key Remapping via as:*

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>;

// { getName: () => string; getAge: () => number; getLocation: () => string; }

You can filter out keys by producing never via a conditional type:

type RemoveKindField<Type> = {

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

};

interface Circle {

kind: "circle";

radius: number;

}

type KindlessCircle = RemoveKindField<Circle>; // {radius: number;}

You can map over arbitrary unions, not just unions of string | number | symbol, but unions of any type:

type EventConfig<Events extends { kind: string }> = {

[E in Events as E["kind"]]: (event: E) => void;

}

type SquareEvent = { kind: "square", x: number, y: number };

type CircleEvent = { kind: "circle", radius: number };

type Config = EventConfig<SquareEvent | CircleEvent>

// { square: (event: SquareEvent) => void; circle: (event: CircleEvent) => void; }

Mapped types work well with other features in this type manipulation section, for example here is a mapped type using a conditional type which returns either a true or false depending on whether an object has the property pii set to the literal true

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>; //{id: false; name:true; }

*Template Literal Types:*

type EmailLocaleIDs = "welcome\_email" | "email\_heading";

type FooterLocaleIDs = "footer\_title" | "footer\_sendoff";

type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}\_id`;

// "welcome\_email\_id" | "email\_heading\_id" | "footer\_title\_id" | "footer\_sendoff\_id"

type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}\_id`;

type Lang = "en" | "ja" | "pt";

type LocaleMessageIDs = `${Lang}\_${AllLocaleIDs}`;

// "en\_welcome\_email\_id" | "en\_email\_heading\_id" | "en\_footer\_title\_id" | "en\_footer\_sendoff\_id" | "ja\_welcome\_email\_id" | "ja\_email\_heading\_id" | "ja\_footer\_title\_id" | "ja\_footer\_sendoff\_id" | "pt\_welcome\_email\_id" | "pt\_email\_heading\_id" | "pt\_footer\_title\_id" | "pt\_footer\_sendoff\_id"

*Generic Classes:*

Classes, much like interfaces, can be generic. When a generic class is instantiated with new, its type parameters are inferred the same way as in a function call

class Box<Type> {

contents: Type;

constructor(value: Type) {

this.contents = value;

}

}

const b = new Box("hello!"); // Box<string>

Static members cannot reference class type parameters.

*this parameters:*

In a method or function definition, an initial parameter named this has special meaning in TypeScript. These parameters are erased during compilation:

function fn(this: SomeType, x: number) {/\* ... \*/}

You can also use this in a parameter type annotation:

class Box {

content: string = "";

sameAs(other: this) {

return other.content === this.content;

}

}

This is different from writing other: Box — if you have a derived class, its sameAs method will now only accept other instances of that same derived class:

class Box {

content: string = "";

sameAs(other: this) {

return other.content === this.content;

}

}

class DerivedBox extends Box {

otherContent: string = "?";

}

const base = new Box();

const derived = new DerivedBox();

derived.sameAs(base); // Error

*this -based type guards:*

You can use this is Type in the return position for methods in classes and interfaces. When mixed with a type narrowing the type of the target object would be narrowed to the specified Type.

class FileSystemObject {

isFile(): this is FileRep {

return this instanceof FileRep;

}

}

*abstract Classes and Members:*

Classes, methods, and fields in TypeScript may be abstract. An abstract method or abstract field is one that hasn’t had an implementation provided. These members must exist inside an abstract class, which cannot be directly instantiated. The role of abstract classes is to serve as a base class for subclasses which do implement all the abstract members. When a class doesn’t have any abstract members, it is said to be concrete. Let’s look at an example:

abstract class Base {

abstract getName(): string;

printName() {

console.log("Hello, " + this.getName());

}

}

const b = new Base(); // Cannot create an instance of an abstract class.

*Relationships Between Classes:*

In most cases, classes in TypeScript are compared structurally, the same as other types. For example, these two classes can be used in place of each other because they’re identical:

class Point1 {

x = 0;

y = 0;

}

class Point2 {

x = 0;

y = 0;

}

const p: Point1 = new Point2(); // OK

Similarly, subtype relationships between classes exist even if there’s no explicit inheritance:

class Person {

name: string;

age: number;

}

class Employee {

name: string;

age: number;

salary: number;

}

const p: Person = new Employee();  // OK

class Empty {}

function fn(x: Empty) {

// can't do anything with 'x', so I won't

}

// All OK!

fn(window);

fn({});

fn(fn);

*Utility Types:*

1. *Awaited<Type>:*

This type is meant to model operations like await in async functions, or the .then() method on Promises - specifically, the way that they recursively unwrap Promises.

type A = Awaited<Promise<string>>; //string

type B = Awaited<Promise<Promise<number>>>; //number

type C = Awaited<boolean | Promise<number>>; // number | boolean

1. *Partial<Type>:*

Constructs a type with all properties of Type set to optional. This utility will return a type that represents all subsets of a given type.

1. *Required<Type>:*

Constructs a type consisting of all properties of Type set to required. The opposite of Partial.

1. *Readonly<Type>:*

Constructs a type with all properties of Type set to readonly, meaning the properties of the constructed type cannot be reassigned.

1. *Record<Keys, Type>:*

Constructs an object type whose property keys are Keys and whose property values are Type. This utility can be used to map the properties of a type to another type.

interface CatInfo {

age: number;

breed: string;

}

type CatName = "miffy" | "boris" | "mordred";

const cats: Record<CatName, CatInfo> = {

miffy: { age: 10, breed: "Persian" },

boris: { age: 5, breed: "Maine Coon" },

mordred: { age: 16, breed: "British Shorthair" },

};

1. *Pick<Type, Keys>:*

Constructs a type by picking the set of properties Keys (string literal or union of string literals) from Type.

interface Todo {

title: string;

description: string;

completed: boolean;

}

type TodoPreview = Pick<Todo, "title" | "completed">;

1. *Omit<Type, Keys>:*

Constructs a type by picking all properties from Type and then removing Keys (string literal or union of string literals). The opposite of Pick.

1. *Exclude<UnionType, ExcludedMembers>:*

Constructs a type by excluding from UnionType all union members that are assignable to ExcludedMembers.

type T0 = Exclude<"a" | "b" | "c", "a">; // b | c

type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // c

type T2 = Exclude<string | number | (() => void), Function> // string | number

1. *Extract<Type, Union>:*

Constructs a type by extracting from Type all union members that are assignable to Union.

type T0 = Extract<"a" | "b" | "c", "a" | "f">; // a

type T1 = Extract<string | number |(() => void), Function>; // ()=>void

1. *NonNullable<Type>:*

Constructs a type by excluding null and undefined from Type.

type T0 = NonNullable<string | number | undefined>; // string | number

type T1 = NonNullable<string[] | null | undefined>; // string[]

1. *Parameters<Type>:*

Constructs a tuple type from the types used in the parameters of a function type Type.

declare function f1(arg: { a: number; b: string }): void;

type T0 = Parameters<() => string>; // []

type T1 = Parameters<(s: string) => void>; // [s: string]

type T2 = Parameters<<T>(arg: T) => T>; // [arg: unknown]

type T3 = Parameters<typeof f1>; [arg: { a:number; b: string;}]

type T4 = Parameters<any>; // unknown[]

type T5 = Parameters<never>; // never

type T6 = Parameters<string>; // error never

type T7 = Parameters<Function>;// error never

1. *ConstructorParameters<Type>:*

Constructs a tuple or array type from the types of a constructor function type. It produces a tuple type with all the parameter types (or the type never if Type is not a function).

type T0 = ConstructorParameters<ErrorConstructor>;

// [message?: string]

type T1 = ConstructorParameters<FunctionConstructor>;

// string[]

type T2 = ConstructorParameters<RegExpConstructor>;

// [pattern: string | RegExp, flags?: string]

class C {

constructor(a: number, b: string) {}

}

type T3 = ConstructorParameters<typeof C>;

//[a:number, b: string]

type T4 = ConstructorParameters<any>; //unknown[]

type T5 = ConstructorParameters<Function>; // error never

1. *ReturnType<Type>:*

Constructs a type consisting of the return type of function Type.

declare function f1(): { a: number; b: string };

type T0 = ReturnType<() => string>; // string

type T1 = ReturnType<(s: string) => void>; // void

type T2 = ReturnType<<T>() => T>; // unknown

type T3 = ReturnType<<T extends U,U extends number[]>()=>T>

// number[]

type T4 = ReturnType<typeof f1>;// { a:number; b:string }

type T5 = ReturnType<any>; // any

type T6 = ReturnType<never>; // never

type T7 = ReturnType<string>; // error any

type T8 = ReturnType<Function>; // error any

1. *InstanceType<Type>:*

Constructs a type consisting of the instance type of a constructor function in Type.

class C {

x = 0;

y = 0;

}

type T0 = InstanceType<typeof C>; // C

type T1 = InstanceType<any>; // any

type T2 = InstanceType<never>; // never

type T3 = InstanceType<string>; //error any

type T4 = InstanceType<Function>; //error any

1. *ThisParameterType<Type>:*

Extracts the type of the this parameter for a function type, or unknown if the function type has no this parameter.

function toHex(this: Number) {

return this.toString(16);

}

function numberToString(n: ThisParameterType<typeof toHex>) {

return toHex.apply(n);

}

1. *OmitThisParameter<Type>:*

Removes the this parameter from Type. If Type has no explicitly declared this parameter, the result is simply Type. Otherwise, a new function type with no this parameter is created from Type. Generics are erased and only the last overload signature is propagated into the new function type.

function toHex(this: Number) {

return this.toString(16);

}

const fiveToHex: OmitThisParameter<typeof toHex> = toHex.bind(5); //()=>string

*Intrinsic String Manipulation Types:*

* Uppercase<StringType>
* Lowercase<StringType>
* Capitalize<StringType>
* Uncapitalize<StringType>