

TypeScript 3.2 is now available. Download ([/#download-links](#)) our latest version today!

Documentation ▼

Advanced Types

Intersection Types

An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, `Person & Serializable & Loggable` is a `Person` *and* `Serializable` *and* `Loggable`. That means an object of this type will have all members of all three types.

You will mostly see intersection types used for mixins and other concepts that don't fit in the classic object-oriented mold. (There are a lot of these in JavaScript!) Here's a simple example that shows how to create a mixin:

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}

interface Loggable {
    log(): void;
}

class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}

var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

Union Types

Union types are closely related to intersection types, but they are used very differently. Occasionally, you'll run into a library that expects a parameter to be either a `number` or a `string`. For instance, take the following function:

```

/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"

```

The problem with `padLeft` is that its `padding` parameter is typed as `any`. That means that we can call it with an argument that's neither a `number` nor a `string`, but TypeScript will be okay with it.

```
let indentedString = padLeft("Hello world", true); // passes at compile time, fails at runtime.
```

In traditional object-oriented code, we might abstract over the two types by creating a hierarchy of types. While this is much more explicit, it's also a little bit overkill. One of the nice things about the original version of `padLeft` was that we were able to just pass in primitives. That meant that usage was simple and concise. This new approach also wouldn't help if we were just trying to use a function that already exists elsewhere.

Instead of `any`, we can use a *union type* for the `padding` parameter:

```

/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
  // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation

```

A union type describes a value that can be one of several types. We use the vertical bar (`|`) to separate each type, so `number | string | boolean` is the type of a value that can be a `number`, a `string`, or a `boolean`.

If we have a value that has a union type, we can only access members that are common to all types in the union.

```

interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
pet.layEggs(); // okay
pet.swim();    // errors

```

Union types can be a bit tricky here, but it just takes a bit of intuition to get used to. If a value has the type `A | B`, we only know for *certain* that it has members that both `A` and `B` have. In this example, `Bird` has a member named `fly`. We can't be sure whether a variable typed as `Bird | Fish` has a `fly` method. If the variable is really a `Fish` at runtime, then calling `pet.fly()` will fail.

Type Guards and Differentiating Types

Union types are useful for modeling situations when values can overlap in the types they can take on. What happens when we need to know specifically whether we have a `Fish`? A common idiom in JavaScript to differentiate between two possible values is to check for the presence of a member. As we mentioned, you can only access members that are guaranteed to be in all the constituents of a union type.

```

let pet = getSmallPet();

// Each of these property accesses will cause an error
if (pet.swim) {
    pet.swim();
}
else if (pet.fly) {
    pet.fly();
}

```

To get the same code working, we'll need to use a type assertion:

```

let pet = getSmallPet();

if ((<Fish>pet).swim) {
    (<Fish>pet).swim();
}
else {
    (<Bird>pet).fly();
}

```

User-Defined Type Guards

Notice that we had to use type assertions several times. It would be much better if once we performed the check, we could know the type of `pet` within each branch.

It just so happens that TypeScript has something called a *type guard*. A type guard is some expression that performs a runtime check that guarantees the type in some scope. To define a 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 (<Fish>pet).swim !== undefined;
}
```

`pet is Fish` is our type predicate in this example. A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

Any time `isFish` is called with some variable, TypeScript will *narrow* that variable to that specific type if the original type is compatible.

```
// Both calls to 'swim' and 'fly' are now okay.

if (isFish(pet)) {
    pet.swim();
}
else {
    pet.fly();
}
```

Notice that TypeScript not only knows that `pet` is a `Fish` in the `if` branch; it also knows that in the `else` branch, you *don't* have a `Fish`, so you must have a `Bird`.

typeof type guards

Let's go back and write the code for the version of `padLeft` that uses union types. We could write it with type predicates as follows:

```
function isNumber(x: any): x is number {
    return typeof x === "number";
}

function isString(x: any): x is string {
    return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
    if (isNumber(padding)) {
        return Array(padding + 1).join(" ") + value;
    }
    if (isString(padding)) {
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
```

However, having to define a function to figure out if a type is a primitive is kind of a pain. Luckily, you don't need to abstract `typeof x === "number"` into its own function because TypeScript will recognize it as a type guard on its own. That means we could just write these checks inline.

```
function padLeft(value: string, padding: string | number) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}
```

These *typeof type guards* are recognized in two different forms: `typeof v === "typename"` and `typeof v !== "typename"`, where "typename" must be "number", "string", "boolean", or "symbol". While TypeScript won't stop you from comparing to other strings, the language won't recognize those expressions as type guards.

instanceof type guards

If you've read about `typeof` type guards and are familiar with the `instanceof` operator in JavaScript, you probably have some idea of what this section is about.

instanceof type guards are a way of narrowing types using their constructor function. For instance, let's borrow our industrial string-padder example from earlier:

```
interface Padder {
  getPaddingString(): string
}

class SpaceRepeatingPadder implements Padder {
  constructor(private numSpaces: number) { }
  getPaddingString() {
    return Array(this.numSpaces + 1).join(" ");
  }
}

class StringPadder implements Padder {
  constructor(private value: string) { }
  getPaddingString() {
    return this.value;
  }
}

function getRandomPadder() {
  return Math.random() < 0.5 ?
    new SpaceRepeatingPadder(4) :
    new StringPadder(" ");
}

// Type is 'SpaceRepeatingPadder | StringPadder'
let padder: Padder = getRandomPadder();

if (padder instanceof SpaceRepeatingPadder) {
  padder; // type narrowed to 'SpaceRepeatingPadder'
}
if (padder instanceof StringPadder) {
  padder; // type narrowed to 'StringPadder'
}
```

The right side of the `instanceof` needs to be a constructor function, and TypeScript will narrow down to:

1. the type of the function's `prototype` property if its type is not any

2. the union of types returned by that type's construct signatures

in that order.

Nullable types

TypeScript has two special types, `null` and `undefined`, that have the values `null` and `undefined` respectively. We mentioned these briefly in the Basic Types section ([./basic-types.html](#)). By default, the type checker considers `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` are valid values of every type. That means it's not possible to *stop* them from being assigned to any type, even when you would like to prevent it. The inventor of `null`, Tony Hoare, calls this his "billion dollar mistake" (https://en.wikipedia.org/wiki/Null_pointer#History).

The `--strictNullChecks` flag fixes this: when you declare a variable, it doesn't automatically include `null` or `undefined`. You can include them explicitly using a union type:

```
let s = "foo";
s = null; // error, 'null' is not assignable to 'string'
let sn: string | null = "bar";
sn = null; // ok

sn = undefined; // error, 'undefined' is not assignable to 'string | null'
```

Note that TypeScript treats `null` and `undefined` differently in order to match JavaScript semantics. `string | null` is a different type than `string | undefined` and `string | undefined | null`.

Optional parameters and properties

With `--strictNullChecks`, an optional parameter automatically adds `| undefined`:

```
function f(x: number, y?: number) {
    return x + (y || 0);
}
f(1, 2);
f(1);
f(1, undefined);
f(1, null); // error, 'null' is not assignable to 'number | undefined'
```

The same is true for optional properties:

```
class C {
    a: number;
    b?: number;
}
let c = new C();
c.a = 12;
c.a = undefined; // error, 'undefined' is not assignable to 'number'
c.b = 13;
c.b = undefined; // ok
c.b = null; // error, 'null' is not assignable to 'number | undefined'
```

Type guards and type assertions

Since nullable types are implemented with a union, you need to use a type guard to get rid of the `null`. Fortunately, this is the same code you'd write in JavaScript:

```
function f(sn: string | null): string {
  if (sn == null) {
    return "default";
  }
  else {
    return sn;
  }
}
```

The `null` elimination is pretty obvious here, but you can use terser operators too:

```
function f(sn: string | null): string {
  return sn || "default";
}
```

In cases where the compiler can't eliminate `null` or `undefined`, you can use the type assertion operator to manually remove them. The syntax is postfix `!: identifier!` removes `null` and `undefined` from the type of `identifier`:

```
function broken(name: string | null): string {
  function postfix(epithet: string) {
    return name.charAt(0) + '. the ' + epithet; // error, 'name' is possibly null
  }
  name = name || "Bob";
  return postfix("great");
}

function fixed(name: string | null): string {
  function postfix(epithet: string) {
    return name!.charAt(0) + '. the ' + epithet; // ok
  }
  name = name || "Bob";
  return postfix("great");
}
```

The example uses a nested function here because the compiler can't eliminate nulls inside a nested function (except immediately-invoked function expressions). That's because it can't track all calls to the nested function, especially if you return it from the outer function. Without knowing where the function is called, it can't know what the type of `name` will be at the time the body executes.

Type Aliases

Type aliases create a new name for a type. Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that you'd otherwise have to write by hand.

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
  if (typeof n === "string") {
    return n;
  }
  else {
    return n();
  }
}
```

Aliasing doesn't actually create a new type - it creates a new *name* to refer to that type. Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.

Just like interfaces, type aliases can also be generic - we can just add type parameters and use them on the right side of the alias declaration:

```
type Container<T> = { value: T };
```

We can also have a type alias refer to itself in a property:

```
type Tree<T> = {  
  value: T;  
  left: Tree<T>;  
  right: Tree<T>;  
}
```

Together with intersection types, we can make some pretty mind-bending types:

```
type LinkedList<T> = T & { next: LinkedList<T> };  
  
interface Person {  
  name: string;  
}  
  
var people: LinkedList<Person>;  
var s = people.name;  
var s = people.next.name;  
var s = people.next.next.name;  
var s = people.next.next.next.name;
```

However, it's not possible for a type alias to appear anywhere else on the right side of the declaration:

```
type Yikes = Array<Yikes>; // error
```

Interfaces vs. Type Aliases

As we mentioned, type aliases can act sort of like interfaces; however, there are some subtle differences.

One difference is that interfaces create a new name that is used everywhere. Type aliases don't create a new name — for instance, error messages won't use the alias name. In the code below, hovering over `interfaced` in an editor will show that it returns an `Interface`, but will show that `aliased` returns object literal type.

```
type Alias = { num: number }  
interface Interface {  
  num: number;  
}  
  
declare function aliased(arg: Alias): Alias;  
declare function interfaced(arg: Interface): Interface;
```

A second more important difference is that type aliases cannot be extended or implemented from (nor can they extend/implement other types). Because an ideal property of software is being open to extension (https://en.wikipedia.org/wiki/Open/closed_principle), you should always use an interface over a type alias if possible.

On the other hand, if you can't express some shape with an interface and you need to use a union or tuple type, type aliases are usually the way to go.

String Literal Types

String literal types allow you to specify the exact value a string must have. In practice string literal types combine nicely with union types, type guards, and type aliases. You can use these features together to get enum-like behavior with strings.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";  
class UIElement {  
  animate(dx: number, dy: number, easing: Easing) {  
    if (easing === "ease-in") {  
      // ...  
    }  
    else if (easing === "ease-out") {  
    }  
    else if (easing === "ease-in-out") {  
    }  
    else {  
      // error! should not pass null or undefined.  
    }  
  }  
}  
  
let button = new UIElement();  
button.animate(0, 0, "ease-in");  
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

You can pass any of the three allowed strings, but any other string will give the error

```
Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" | "ease-out" | "ease-in-out"'
```

String literal types can be used in the same way to distinguish overloads:

```
function createElement(tagName: "img"): HTMLImageElement;  
function createElement(tagName: "input"): HTMLInputElement;  
// ... more overloads ...  
function createElement(tagName: string): Element {  
  // ... code goes here ...  
}
```

Numeric Literal Types

TypeScript also has numeric literal types.

```
function rollDie(): 1 | 2 | 3 | 4 | 5 | 6 {  
  // ...  
}
```

These are seldom written explicitly, they can be useful when narrowing can catch bugs:

```
function foo(x: number) {  
  if (x !== 1 || x !== 2) {  
    // ~~~~~  
    // Operator '!==' cannot be applied to types '1' and '2'.  
  }  
}
```

In other words, `x` must be `1` when it gets compared to `2`, meaning that the above check is making an invalid comparison.

Enum Member Types

As mentioned in our section on enums ([./enums.html#union-enums-and-enum-member-types](#)), enum members have types when every member is literal-initialized.

Much of the time when we talk about “singleton types”, we’re referring to both enum member types as well as numeric/string literal types, though many users will use “singleton types” and “literal types” interchangeably.

Discriminated Unions

You can combine singleton types, union types, type guards, and type aliases to build an advanced pattern called *discriminated unions*, also known as *tagged unions* or *algebraic data types*. Discriminated unions are useful in functional programming. Some languages automatically discriminate unions for you; TypeScript instead builds on JavaScript patterns as they exist today. There are three ingredients:

1. Types that have a common, singleton type property — the *discriminant*.
2. A type alias that takes the union of those types — the *union*.
3. Type guards on the common property.

```
interface Square {  
  kind: "square";  
  size: number;  
}  
interface Rectangle {  
  kind: "rectangle";  
  width: number;  
  height: number;  
}  
interface Circle {  
  kind: "circle";  
  radius: number;  
}
```

First we declare the interfaces we will union. Each interface has a `kind` property with a different string literal type. The `kind` property is called the *discriminant* or *tag*. The other properties are specific to each interface. Notice that the interfaces are currently unrelated. Let’s put them into a union:

```
type Shape = Square | Rectangle | Circle;
```

Now let’s use the discriminated union:

```
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
}
```

Exhaustiveness checking

We would like the compiler to tell us when we don't cover all variants of the discriminated union. For example, if we add `Triangle` to `Shape`, we need to update `area` as well:

```
type Shape = Square | Rectangle | Circle | Triangle;
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
  // should error here - we didn't handle case "triangle"
}
```

There are two ways to do this. The first is to turn on `--strictNullChecks` and specify a return type:

```
function area(s: Shape): number { // error: returns number | undefined
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
  }
}
```

Because the `switch` is no longer exhaustive, TypeScript is aware that the function could sometimes return `undefined`. If you have an explicit return type `number`, then you will get an error that the return type is actually `number | undefined`. However, this method is quite subtle and, besides, `--strictNullChecks` does not always work with old code.

The second method uses the `never` type that the compiler uses to check for exhaustiveness:

```
function assertNever(x: never): never {
  throw new Error("Unexpected object: " + x);
}
function area(s: Shape) {
  switch (s.kind) {
    case "square": return s.size * s.size;
    case "rectangle": return s.height * s.width;
    case "circle": return Math.PI * s.radius ** 2;
    default: return assertNever(s); // error here if there are missing cases
  }
}
```

Here, `assertNever` checks that `s` is of type `never` — the type that's left after all other cases have been removed. If you forget a case, then `s` will have a real type and you will get a type error. This method requires you to define an extra function, but it's much more obvious when you forget it.

Polymorphic `this` types

A polymorphic `this` type represents a type that is the *subtype* of the containing class or interface. This is called *F*-bounded polymorphism. This makes hierarchical fluent interfaces much easier to express, for example. Take a simple calculator that returns `this` after each operation:

```
class BasicCalculator {  
    public constructor(protected value: number = 0) { }  
    public currentValue(): number {  
        return this.value;  
    }  
    public add(operand: number): this {  
        this.value += operand;  
        return this;  
    }  
    public multiply(operand: number): this {  
        this.value *= operand;  
        return this;  
    }  
    // ... other operations go here ...  
}  
  
let v = new BasicCalculator(2)  
    .multiply(5)  
    .add(1)  
    .currentValue();
```

Since the class uses `this` types, you can extend it and the new class can use the old methods with no changes.

```
class ScientificCalculator extends BasicCalculator {  
    public constructor(value = 0) {  
        super(value);  
    }  
    public sin() {  
        this.value = Math.sin(this.value);  
        return this;  
    }  
    // ... other operations go here ...  
}  
  
let v = new ScientificCalculator(2)  
    .multiply(5)  
    .sin()  
    .add(1)  
    .currentValue();
```

Without `this` types, `ScientificCalculator` would not have been able to extend `BasicCalculator` and keep the fluent interface. `multiply` would have returned `BasicCalculator`, which doesn't have the `sin` method. However, with `this` types, `multiply` returns `this`, which is `ScientificCalculator` here.

Index types

With index types, you can get the compiler to check code that uses dynamic property names. For example, a common Javascript pattern is to pick a subset of properties from an object:

```
function pluck(o, names) {
  return names.map(n => o[n]);
}
```

Here's how you would write and use this function in TypeScript, using the **index type query** and **indexed access** operators:

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
  return names.map(n => o[n]);
}

interface Person {
  name: string;
  age: number;
}
let person: Person = {
  name: 'Jarid',
  age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

The compiler checks that `name` is actually a property on `Person`. The example introduces a couple of new type operators. First is `keyof T`, the **index type query operator**. For any type `T`, `keyof T` is the union of known, public property names of `T`. For example:

```
let personProps: keyof Person; // 'name' | 'age'
```

`keyof Person` is completely interchangeable with `'name' | 'age'`. The difference is that if you add another property to `Person`, say `address: string`, then `keyof Person` will automatically update to be `'name' | 'age' | 'address'`. And you can use `keyof` in generic contexts like `pluck`, where you can't possibly know the property names ahead of time. That means the compiler will check that you pass the right set of property names to `pluck`:

```
pluck(person, ['age', 'unknown']); // error, 'unknown' is not in 'name' | 'age'
```

The second operator is `T[K]`, the **indexed access operator**. Here, the type syntax reflects the expression syntax. That means that `person['name']` has the type `Person['name']` — which in our example is just `string`. However, just like index type queries, you can use `T[K]` in a generic context, which is where its real power comes to life. You just have to make sure that the type variable `K` extends `keyof T`. Here's another example with a function named `getProperty`.

```
function getProperty<T, K extends keyof T>(o: T, name: K): T[K] {
  return o[name]; // o[name] is of type T[K]
}
```

In `getProperty`, `o: T` and `name: K`, so that means `o[name]: T[K]`. Once you return the `T[K]` result, the compiler will instantiate the actual type of the key, so the return type of `getProperty` will vary according to which property you request.

```
let name: string = getProperty(person, 'name');
let age: number = getProperty(person, 'age');
let unknown = getProperty(person, 'unknown'); // error, 'unknown' is not in 'name' | 'age'
```

Index types and string index signatures

`keyof` and `T[K]` interact with string index signatures. If you have a type with a string index signature, `keyof T` will just be `string`. And `T[string]` is just the type of the index signature:

```
interface Map<T> {  
    [key: string]: T;  
}  
let keys: keyof Map<number>; // string  
let value: Map<number>['foo']; // number
```

Mapped types

A common task is to take an existing type and make each of its properties optional:

```
interface PersonPartial {  
    name?: string;  
    age?: number;  
}
```

Or we might want a readonly version:

```
interface PersonReadOnly {  
    readonly name: string;  
    readonly age: number;  
}
```

This happens often enough in Javascript that TypeScript provides a way to create new types based on old types — **mapped types**. In a mapped type, the new type transforms each property in the old type in the same way. For example, you can make all properties of a type `readonly` or optional. Here are a couple of examples:

```
type Readonly<T> = {  
    readonly [P in keyof T]: T[P];  
}  
type Partial<T> = {  
    [P in keyof T]?: T[P];  
}
```

And to use it:

```
type PersonPartial = Partial<Person>;  
type ReadonlyPerson = Readonly<Person>;
```

Let's take a look at the simplest mapped type and its parts:

```
type Keys = 'option1' | 'option2';  
type Flags = { [K in Keys]: boolean };
```

The syntax resembles the syntax for index signatures with a `for ... in` inside. There are three parts:

1. The type variable `K`, which gets bound to each property in turn.
2. The string literal union `Keys`, which contains the names of properties to iterate over.
3. The resulting type of the property.

In this simple example, `Keys` is a hard-coded list of property names and the property type is always `boolean`, so this mapped type is equivalent to writing:

```
type Flags = {
  option1: boolean;
  option2: boolean;
}
```

Real applications, however, look like `Readonly` or `Partial` above. They're based on some existing type, and they transform the properties in some way. That's where `keyof` and indexed access types come in:

```
type NullablePerson = { [P in keyof Person]: Person[P] | null }
type PartialPerson = { [P in keyof Person]?: Person[P] }
```

But it's more useful to have a general version.

```
type Nullable<T> = { [P in keyof T]: T[P] | null }
type Partial<T> = { [P in keyof T]?: T[P] }
```

In these examples, the properties list is `keyof T` and the resulting type is some variant of `T[P]`. This is a good template for any general use of mapped types. That's because this kind of transformation is homomorphic (<https://en.wikipedia.org/wiki/Homomorphism>), which means that the mapping applies only to properties of `T` and no others. The compiler knows that it can copy all the existing property modifiers before adding any new ones. For example, if `Person.name` was `readonly`, `Partial<Person>.name` would be `readonly` and `optional`.

Here's one more example, in which `T[P]` is wrapped in a `Proxy<T>` class:

```
type Proxy<T> = {
  get(): T;
  set(value: T): void;
}
type Proxify<T> = {
  [P in keyof T]: Proxy<T[P]>;
}
function proxify<T>(o: T): Proxify<T> {
  // ... wrap proxies ...
}
let proxyProps = proxify(props);
```

Note that `Readonly<T>` and `Partial<T>` are so useful, they are included in TypeScript's standard library along with `Pick` and `Record`:

```
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
}
type Record<K extends string, T> = {
  [P in K]: T;
}
```

`Readonly`, `Partial` and `Pick` are homomorphic whereas `Record` is not. One clue that `Record` is not homomorphic is that it doesn't take an input type to copy properties from:

```
type ThreeStringProps = Record<'prop1' | 'prop2' | 'prop3', string>
```

Non-homomorphic types are essentially creating new properties, so they can't copy property modifiers from anywhere.

Inference from mapped types

Now that you know how to wrap the properties of a type, the next thing you'll want to do is unwrap them. Fortunately, that's pretty easy:

```
function unproxify<T>(t: Proxify<T>): T {  
    let result = {} as T;  
    for (const k in t) {  
        result[k] = t[k].get();  
    }  
    return result;  
}  
  
let originalProps = unproxify(proxyProps);
```

Note that this unwrapping inference only works on homomorphic mapped types. If the mapped type is not homomorphic you'll have to give an explicit type parameter to your unwrapping function.

Conditional Types

TypeScript 2.8 introduces *conditional types* which add the ability to express non-uniform type mappings. A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. When `T` or `U` contains type variables, whether to resolve to `X` or `Y`, or to defer, is determined by whether or not the type system has enough information to conclude that `T` is always assignable to `U`.

As an example of some types that are immediately resolved, we can take a look at the following example:

```
declare function f<T extends boolean>(x: T): T extends true ? string : number;  
  
// Type is 'string | number'  
let x = f(Math.random() < 0.5)
```

Another example would be the `TypeName` type alias, which uses nested conditional types:


```

type TypeName<T> =
  T extends string ? "string" :
  T extends number ? "number" :
  T extends boolean ? "boolean" :
  T extends undefined ? "undefined" :
  T extends Function ? "function" :
  "object";

type T0 = TypeName<string>; // "string"
type T1 = TypeName<"a">; // "string"
type T2 = TypeName<true>; // "boolean"
type T3 = TypeName<() => void>; // "function"
type T4 = TypeName<string[]>; // "object"

```

But as an example of a place where conditional types are deferred - where they stick around instead of picking a branch - would be in the following:

```

interface Foo {
  propA: boolean;
  propB: boolean;
}

declare function f<T>(x: T): T extends Foo ? string : number;

function foo<U>(x: U) {
  // Has type 'U extends Foo ? string : number'
  let a = f(x);

  // This assignment is allowed though!
  let b: string | number = a;
}

```

In the above, the variable `a` has a conditional type that hasn't yet chosen a branch. When another piece of code ends up calling `foo`, it will substitute in `U` with some other type, and TypeScript will re-evaluate the conditional type, deciding whether it can actually pick a branch.

In the meantime, we can assign a conditional type to any other target type as long as each branch of the conditional is assignable to that target. So in our example above we were able to assign `U extends Foo ? string : number` to `string | number` since no matter what the conditional evaluates to, it's known to be either `string` or `number`.

Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called *distributive conditional types*. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of `T extends U ? X : Y` with the type argument `A | B | C` for `T` is resolved as `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`.

Example

```

type T10 = TypeName<string | (() => void)>; // "string" | "function"
type T12 = TypeName<string | string[] | undefined>; // "string" | "object" | "undefined"
type T11 = TypeName<string[] | number[]>; // "object"

```

In instantiations of a distributive conditional type `T extends U ? X : Y`, references to `T` within the conditional type are resolved to individual constituents of the union type (i.e. `T` refers to the individual constituents *after* the conditional type is distributed over the union type). Furthermore, references to `T` within `X` have an additional type parameter constraint `U` (i.e. `T` is considered assignable to

U within X).

Example

```
type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>;

type T20 = Boxed<string>; // BoxedValue<string>;
type T21 = Boxed<number[]>; // BoxedArray<number>;
type T22 = Boxed<string | number[]>; // BoxedValue<string> | BoxedArray<number>;
```

Notice that `T` has the additional constraint `any[]` within the true branch of `Boxed<T>` and it is therefore possible to refer to the element type of the array as `T[number]`. Also, notice how the conditional type is distributed over the union type in the last example.

The distributive property of conditional types can conveniently be used to *filter* union types:

```
type Diff<T, U> = T extends U ? never : T; // Remove types from T that are assignable to U
type Filter<T, U> = T extends U ? T : never; // Remove types from T that are not assignable to U

type T30 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T31 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"
type T32 = Diff<string | number | (() => void), Function>; // string | number
type T33 = Filter<string | number | (() => void), Function>; // () => void

type NonNullable<T> = Diff<T, null | undefined>; // Remove null and undefined from T

type T34 = NonNullable<string | number | undefined>; // string | number
type T35 = NonNullable<string | string[] | null | undefined>; // string | string[]

function f1<T>(x: T, y: NonNullable<T>) {
  x = y; // Ok
  y = x; // Error
}

function f2<T extends string | undefined>(x: T, y: NonNullable<T>) {
  x = y; // Ok
  y = x; // Error
  let s1: string = x; // Error
  let s2: string = y; // Ok
}
```

Conditional types are particularly useful when combined with mapped types:

```

type FunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? K : never }[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = { [K in keyof T]: T[K] extends Function ? never : K }[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {
  id: number;
  name: string;
  subparts: Part[];
  updatePart(newName: string): void;
}

type T40 = FunctionPropertyNames<Part>; // "updatePart"
type T41 = NonFunctionPropertyNames<Part>; // "id" | "name" | "subparts"
type T42 = FunctionProperties<Part>; // { updatePart(newName: string): void }
type T43 = NonFunctionProperties<Part>; // { id: number, name: string, subparts: Part[] }

```

Similar to union and intersection types, conditional types are not permitted to reference themselves recursively. For example the following is an error.

Example

```

type ElementType<T> = T extends any[] ? ElementType<T[number]> : T; // Error

```

Type inference in conditional types

Within the `extends` clause of a conditional type, it is now possible to have `infer` declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple `infer` locations for the same type variable.

For example, the following extracts the return type of a function type:

```

type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

```

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

```

type Unpacked<T> =
  T extends (infer U)[] ? U :
  T extends (...args: any[]) => infer U ? U :
  T extends Promise<infer U> ? U :
  T;

type T0 = Unpacked<string>; // string
type T1 = Unpacked<string[]>; // string
type T2 = Unpacked<() => string>; // string
type T3 = Unpacked<Promise<string>>; // string
type T4 = Unpacked<Promise<string>[]>; // Promise<string>
type T5 = Unpacked<Unpacked<Promise<string>[]>>; // string

```

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

```
type Foo<T> = T extends { a: infer U, b: infer U } ? U : never;
type T10 = Foo<{ a: string, b: string }>; // string
type T11 = Foo<{ a: string, b: number }>; // string | number
```

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

```
type Bar<T> = T extends { a: (x: infer U) => void, b: (x: infer U) => void } ? U : never;
type T20 = Bar<{ a: (x: string) => void, b: (x: string) => void }>; // string
type T21 = Bar<{ a: (x: string) => void, b: (x: number) => void }>; // string & number
```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;
type T30 = ReturnType<typeof foo>; // string | number
```

It is not possible to use `infer` declarations in constraint clauses for regular type parameters:

```
type ReturnType<T extends (...args: any[]) => infer R> = R; // Error, not supported
```

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

```
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R ? R : any;
```

Predefined conditional types

TypeScript 2.8 adds several predefined conditional types to `lib.d.ts`:

- `Exclude<T, U>` – Exclude from `T` those types that are assignable to `U`.
- `Extract<T, U>` – Extract from `T` those types that are assignable to `U`.
- `Nullable<T>` – Exclude `null` and `undefined` from `T`.
- `ReturnType<T>` – Obtain the return type of a function type.
- `InstanceType<T>` – Obtain the instance type of a constructor function type.

Example

```

type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T01 = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"

type T02 = Exclude<string | number | (() => void), Function>; // string | number
type T03 = Extract<string | number | (() => void), Function>; // () => void

type T04 = NonNullable<string | number | undefined>; // string | number
type T05 = NonNullable<() => string | string[] | null | undefined>; // () => string | string
[]

function f1(s: string) {
    return { a: 1, b: s };
}

class C {
    x = 0;
    y = 0;
}


type T10 = ReturnType<() => string>; // string
type T11 = ReturnType<(s: string) => void>; // void
type T12 = ReturnType<(<T>() => T)>; // {}
type T13 = ReturnType<(<T extends U, U extends number[]>() => T)>; // number[]
type T14 = ReturnType<typeof f1>; // { a: number, b: string }
type T15 = ReturnType<any>; // any
type T16 = ReturnType<never>; // any
type T17 = ReturnType<string>; // Error
type T18 = ReturnType<Function>; // Error

type T20 = InstanceType<typeof C>; // C
type T21 = InstanceType<any>; // any
type T22 = InstanceType<never>; // any
type T23 = InstanceType<string>; // Error
type T24 = InstanceType<Function>; // Error

```

Note: The `Exclude` type is a proper implementation of the `Diff` type suggested here (<https://github.com/Microsoft/TypeScript/issues/12215#issuecomment-307871458>). We've used the name `Exclude` to avoid breaking existing code that defines a `Diff`, plus we feel that name better conveys the semantics of the type. We did not include the `Omit<T, K>` type because it is trivially written as `Pick<T, Exclude<keyof T, K>>`.

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft