Chapter 6. Advanced Types

TypeScript has a world-class type system that supports powerful type-level programming features that might make even the crotchetiest Haskell programmer jealous. As you by now know, that type system isn't just incredibly expressive, but also easy to use, and makes declaring type constraints and relationships simple, terse, and most of the time, inferred.

We need such an expressive and unusual type system because JavaScript is so dynamic. Modeling things like prototypes, dynamically bound `this`, function overloads, and always-changing objects requires a rich type system and a utility belt of type operators that would make Batman do a double-take.

I'll start this chapter with a deep dive into subtyping, assignability, variance, and widening in TypeScript, giving more definition to the intuitions you've been developing over the last several chapters. I'll then cover TypeScript's control-flow-based typechecking features in more detail, including refinement and totality, and continue with some advanced type-level programming features: keying into and mapping over object types, using conditional types, defining your own type guards, and escape hatches like type assertions and definite assignment assertions. Finally, I'll cover advanced patterns for squeezing more safety out of your types: the companion object pattern, improving inference for tuple types, simulating nominal types, and safely extending the prototype.

Relationships Between Types

Let's begin by taking a closer look at type relations in TypeScript.

Subtypes and Supertypes

We talked a little about assignability in "Talking About Types". Now that you've seen most of the types TypeScript has to offer we can dive deeper, starting from the top: what's a subtype?

SUBTYPE

If you have two types `A` and `B`, and `B` is a subtype of `A`, then you can safely use a `B` anywhere an `A` is required (Figure 6-1).

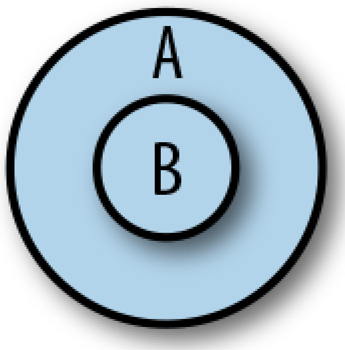

Figure 6-1. B is a subtype of A

If you look back at Figure 3-1 at the very beginning of Chapter 3, you'll see what the subtype relations built into TypeScript are. For example:

- Array is a subtype of Object.
- Tuple is a subtype of Array.
- Everything is a subtype of `any`.
- `never` is a subtype of everything.
- If you have a class `Bird` that extends `Animal`, then `Bird` is a subtype of `Animal`.

From the definition I just gave for subtype, that means:

- Anywhere you need an Object you can also use an Array.
- Anywhere you need an Array you can also use a Tuple.
- Anywhere you need an `any` you can also use an Object.
- You can use a `never` anywhere.
- Anywhere you need an `Animal` you can also use a `Bird`.

As you might have guessed, a supertype is the opposite of a subtype.

SUPERTYPE

If you have two types `A` and `B`, and `B` is a supertype of `A`, then you can safely use an `A` anywhere a `B` is required (Figure 6-2).

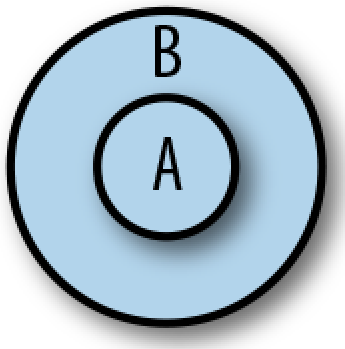

Figure 6-2. B is a supertype of A

Again from the flowchart in Figure 3-1:

- Array is a supertype of Tuple.
- Object is a supertype of Array.
- Any is a supertype of everything.
- Never is a supertype of nothing.
- `Animal` is a supertype of `Bird` .

This is just the opposite of how subtypes work, and nothing more.

Variance

For most types it's pretty easy to intuit whether or not some type `A` is a subtype of another type `B` . For simple types like `number` , `string` , and so on, you can just look them up in the flowchart in Figure 3-1, or reason through it (" `number` is contained in the union `number | string` , so it must be a subtype of it").

But for parameterized (generic) types and other more complex types, it gets more complicated. Consider these cases:

- When is `Array<A>` a subtype of `Array<B>` ?
- When is a shape `A` a subtype of another shape `B` ?
- When is a function `(a: A) => B` a subtype of another function `(c: C) => D` ?

Subtyping rules for types that contain other types (i.e., things with type parameters like `Array<A>` , shapes with fields like `{a: number}` , or functions like `(a: A) => B`) are harder to reason about, and the answers aren't as clear-cut. In fact, subtyping rules for these kinds of complex types are a big point of disagreement among programming languages—almost no two languages are alike!

To make the following rules easier to read, I’m going to introduce a few pieces of syntax that let us talk about types a little more precisely and tersely. This syntax is not valid TypeScript; it’s just a way for you and me to share a common language when we talk about types. And don’t worry, I swear the syntax isn’t math:

- `A <: B` means " `A` is a subtype of or the same as the type `B` .”
- `A >: B` means " `A` is a supertype of or the same as the type `B` .”

Shape and array variance

To get some intuition for why exactly languages disagree on subtyping rules for complex types, let me take you through an example complex type: shapes. Say you have a shape describing a user in your application. You might represent it with a pair of types that look something like this:

```
// An existing user that we got from the server
type ExistingUser = {
  id: number
  name: string
}

// A new user that hasn't been saved to the server yet
type NewUser = {
  name: string
}
```

Now suppose an intern at your company is tasked with writing some code to delete a user. They start it like this:

```
function deleteUser(user: {id?: number, name: string})
  delete user.id
}

let existingUser: ExistingUser = {
  id: 123456,
  name: 'Ima User'
}
```

```
deleteUser(existingUser)
```

`deleteUser` takes an object of type `{id?: number, name: string}`, and it's passed an `existingUser` of type `{id: number, name: string}`. Notice that the type of the `id` property (`number`) is a *subtype* of the expected type (`number | undefined`). Therefore the entire object `{id: number, name: string}` is a subtype of `{id?: number, name: string}`, so TypeScript lets it fly.

Do you see the safety issue here? It's a subtle one: after passing an `ExistingUser` to `deleteUser`, TypeScript doesn't know that the user's `id` has been deleted, so if we read `existingUser.id` after deleting it with `deleteUser(existingUser)`, TypeScript still thinks `existingUser.id` is of type `number`!

Clearly, using an object type in a place where something expects its supertype can be unsafe. So why does TypeScript allow it? In general, TypeScript is not designed to be perfectly safe; instead, its type system tries to strike a balance between catching real mistakes and being easy to use, without you needing to get a degree in programming language theory to understand why something is an error. This specific case of unsafety is a practical one: since destructive updates (like deleting a property) are relatively rare in practice, TypeScript is lax and lets you assign an object to a place where its supertype is expected.

What about the opposite direction—can you assign an object to a place where its subtype is expected?

Let's add a new type for a legacy user, then delete a user of that type (imagine you're adding types to code your coworker wrote before you started using TypeScript):

```
type LegacyUser = {
  id?: number | string
  name: string
}

let legacyUser: LegacyUser = {
  id: '793331',
  name: 'Xin Yang'
}
```

```
deleteUser(legacyUser) // Error TS2345: Argument of typ
                       // assignable to parameter of ty
                       // undefined, name: string}'. Ty
                       // assignable to type 'number |
```

When we pass a shape with a property whose type is a supertype of the expected type, TypeScript complains. That's because `id` is a `string | number | undefined`, and `deleteUser` only handles the case of an `id` that's a `number | undefined`.

TypeScript's behavior is as follows: if you expect a shape, you can also pass a type with property types that are `<:` their expected types, but you cannot pass a shape with property types that are supertypes of their expected types. When talking about types, we say that TypeScript shapes (objects and classes) are *covariant* in their property types. That is, for an object `A` to be assignable to an object `B`, each of its properties must be `<:` its corresponding property in `B`.

More generally, covariance is just one of four sorts of variance:

Invariance

You want exactly a `T`.

Covariance

You want a `<:T`.

Contravariance

You want a `>:T`.

Bivariance

You're OK with either `<:T` or `>:T`.

In TypeScript, every complex type is covariant in its members—objects, classes, arrays, and function return types—with one exception: function parameter types, which are *contravariant*.

NOTE

Not all languages make this same design decision. In some languages objects are *invariant* in their property types, because as we saw, covariant property types can lead to unsafe behavior. Some languages have different rules for mutable and immutable objects (try to reason through it yourself!). Some languages—like Scala, Kotlin, and Flow—even have explicit syntax for programmers to specify variance for their own data types.

When designing TypeScript, its authors opted for a balance between ease of use and safety. When you make objects invariant in their property types, even though it's safer, it can make a type system tedious to use because you end up banning things that are safe in practice (e.g., if we didn't `delete` the `id` in `deleteUser` , then it would have been perfectly safe to pass in an object that's a supertype of the expected type).

Function variance

Let's start with a few examples.

A function `A` is a subtype of function `B` if `A` has the same or lower arity (number of parameters) than `B` and:

1. `A` 's `this` type either isn't specified, or is `>:` `B` 's `this` type.
2. Each of `A` 's parameters is `>:` its corresponding parameter in `B` .
3. `A` 's return type is `<:` `B` 's return type.

Read that over a few times, and make sure you understand what each rule means. You might have noticed that for a function `A` to be a subtype of function `B` , we say that its `this` type and parameters must be `>:` their counterparts in `B` , while its return type has to be `<:` ! Why does the direction flip like that? Why isn't it simply `<:` for each component (`this` type, parameter types, and return type), like it is for objects, arrays, unions, and so on?

To answer this question, let's derive it ourselves. We'll start by defining three types (we're going to use a `class` for clarity, but this works for any choice of types where `A <: B <: C`):

```
class Animal {}
class Bird extends Animal {
  chirp() {}
}
class Crow extends Bird {
  caw() {}
}
```

In this example, `Crow` is a subtype of `Bird`, which is a subtype of `Animal`. That is, `Crow <: Bird <: Animal`.

Now, let's define a function that takes a `Bird`, and makes it chirp:

```
function chirp(bird: Bird): Bird {
  bird.chirp()
  return bird
}
```

So far, so good. What kinds of things does TypeScript let you pass into `chirp`?

```
chirp(new Animal) // Error TS2345: Argument of type 'Ar
chirp(new Bird)   // to parameter of type 'Bird'.
chirp(new Crow)
```

You can pass an instance of `Bird` (because that's what `chirp`'s parameter `bird`'s type is) or an instance of `Crow` (because it's a subtype of `Bird`). Great: passing in a subtype works as expected.

Let's make a new function. This time, its parameter will be a *function*:

```
function clone(f: (b: Bird) => Bird): void {
  // ...
}
```

`clone` needs a function `f` that takes a `Bird` and returns a `Bird`. What types of functions can you safely pass for `f`? Clearly you can pass a function that takes a `Bird` and returns a `Bird`:

```
function birdToBird(b: Bird): Bird {
  // ...
}
clone(birdToBird) // OK
```

What about a function that takes a `Bird` and returns a `Crow`, or an `Animal`?

```
function birdToCrow(d: Bird): Crow {
  // ...
}
clone(birdToCrow) // OK

function birdToAnimal(d: Bird): Animal {
  // ...
}
clone(birdToAnimal) // Error TS2345: Argument of type '
                    // not assignable to parameter of t
                    // Type 'Animal' is not assignable
```

`birdToCrow` works as expected, but `birdToAnimal` gives us an error. Why? Imagine that `clone`'s implementation looks like this:

```
function clone(f: (b: Bird) => Bird): void {
  let parent = new Bird
  let babyBird = f(parent)
  babyBird.chirp()
}
```

If we passed to our `clone` function an `f` that returned an `Animal`, then we couldn't call `.chirp` on it! So TypeScript has to make sure, at compile time, that the function we passed in returns *at least* a `Bird`.

We say that functions are *covariant* in their return types, which is a fancy way of saying that for a function to be a subtype of another function, its return type has to be `<:` the other function's return type.

OK, what about parameter types?

```
function animalToBird(a: Animal): Bird {
  // ...
}
clone(animalToBird) // OK

function crowToBird(c: Crow): Bird {
  // ...
```

```
}
clone(crowToBird) // Error TS2345: Argument of type '(c
                  // assignable to parameter of type '(
```

For a function to be assignable to another function, its parameter types (including `this`) all have to be `>:` their corresponding parameter types in the other function. To see why, think about how a user might have implemented `crowToBird` before passing it into `clone` . What if they did this?

```
function crowToBird(c: Crow): Bird {
  c.caw()
  return new Bird
}
```

Now if `clone` called `crowToBird` with a `new Bird` , we'd get an exception because `.caw` is only defined on `Crow` s, not on all `Bird` s.

This means functions are *contravariant* in their parameter and `this` types. That is, for a function to be a subtype of another function, each of its parameters and its `this` type must be `>:` its corresponding parameter in the other function.

Thankfully, you don't have to memorize and recite these rules. Just have them in the back of your mind when your code editor gives you a red squiggly when you pass an incorrectly typed function somewhere, so you know why TypeScript is giving you the error it does.

TSC FLAG: STRICTFUNCTIONTYPES

For legacy reasons, functions in TypeScript are actually covariant in their parameter and `this` types by default. To opt into the safer, contravariant behavior we just explored, be sure to enable the `{"strictFunctionTypes": true}` flag in your *tsconfig.json*.

`strict` mode includes `strictFunctionTypes` , so if you're already using `{"strict": true}` , you're good to go.

Assignability

Subtype and supertype relations are core concepts in any statically typed language. They're also important to understanding how *assignability* works (as a reminder, assignability refers to TypeScript's rules for whether or not you can use a type `A` where another type `B` is required).

When TypeScript wants to answer the question "Is type `A` assignable to type `B`?" it follows a few simple rules. For *non-enum types*—like arrays, booleans, numbers, objects, functions, classes, class instances, and strings, including literal types—`A` is assignable to `B` if either of the following is true:

1. `A <: B`.
2. `A` is `any`.

Rule 1 is just the definition of what a subtype is: if `A` is a subtype of `B`, then wherever you need a `B` you can also use an `A`.

Rule 2 is the exception to rule 1, and is a convenience for interoperating with JavaScript code.

For *enum types* created with the `enum` or `const enum` keywords, a type `A` is assignable to an enum `B` if either of these is true:

1. `A` is a member of enum `B`.
2. `B` has at least one member that's a `number`, and `A` is a `number`.

Rule 1 is exactly the same as for simple types (if `A` is a member of enum `B`, then `A`'s type is `B`, so all we're saying is `B <: B`).

Rule 2 is a convenience for working with enums. As we talked about in "Enums", rule 2 is a big source of unsafety in TypeScript, and this is one reason I suggest throwing the baby out with the bathwater and avoiding enums entirely.

Type Widening

Type widening is key to understanding how TypeScript's type inference works. In general, TypeScript will be lenient when inferring your types, and will err on the side of inferring a more general type rather than the most specific type possible. This makes your life as a programmer easier, and means less time spent quelling the typechecker's complaints.

In Chapter 3, you already saw a few instances of type widening in action. Let's look at a few more examples.

When you declare a variable in a way that allows it to be mutated later (e.g., with `let` or `var`), its type is widened from its literal value to the base type that literal belongs to:

```
let a = 'x'                 // string
let b = 3                   // number
var c = true                // boolean
const d = {x: 3}            // {x: number}

enum E {X, Y, Z}
let e = E.X                 // E
```

Not so for immutable declarations:

```
const a = 'x'               // 'x'
const b = 3                 // 3
const c = true              // true

enum E {X, Y, Z}
const e = E.X               // E.X
```

You can use an explicit type annotation to prevent your type from being widened:

```
let a: 'x' = 'x'            // 'x'
let b: 3 = 3                // 3
var c: true = true          // true
const d: {x: 3} = {x: 3}    // {x: 3}
```

When you reassign a nonwidened type using `let` or `var`, TypeScript widens it for you. To tell TypeScript to keep it narrow, add an explicit type annotation to your original declaration:

```
const a = 'x'             // 'x'
let b = a                 // string

const c: 'x' = 'x'        // 'x'
let d = c                 // 'x'
```

Variables initialized to `null` or `undefined` are widened to `any`:

```
let a = null              // any
a = 3                     // any
a = 'b'                   // any
```

But when a variable initialized to `null` or `undefined` leaves the scope it was declared in, TypeScript assigns it a definite type:

```
function x() {
  let a = null            // any
  a = 3                   // any
  a = 'b'                 // any
  return a
}

x()                       // string
```

The const type

TypeScript comes with a special `const` type that you can use to opt out of type widening a declaration at a time. Use it as a type assertion (read ahead to "Type Assertions"):

```
let a = {x: 3}                // {x: number}
let b: {x: 3}                 // {x: 3}
let c = {x: 3} as const       // {readonly x: 3}
```

`const` opts your type out of widening and recursively marks its members as `readonly`, even for deeply nested data structures:

```
let d = [1, {x: 2}]           // (number | {x: number})
let e = [1, {x: 2}] as const  // readonly [1, {readonly
```

Use `as const` when you want TypeScript to infer your type as narrowly as possible.

Excess property checking

Type widening also comes into the picture when TypeScript checks whether or not one object type is assignable to another object type.

Recall from "Shape and array variance" that object types are covariant in their members. But if TypeScript stuck to this rule without doing any additional checks, it could lead to a problem.

For example, consider an `Options` object you might pass into a class to configure it:

```
type Options = {
  baseURL: string
  cacheSize?: number
  tier?: 'prod' | 'dev'
}

class API {
  constructor(private options: Options) {}
}

new API({
  baseURL: 'https://api.mysite.com',
  tier: 'prod'
})
```

Now, what happens if you misspell an option?

```
new API({
  baseURL: 'https://api.mysite.com',
  tierr: 'prod'      // Error TS2345: Argument of type
})                   // is not assignable to parameter
                     // Object literal may only specify
                     // but 'tierr' does not exist in t
                     // Did you mean to write 'tier'?
```

This is a common bug when working with JavaScript, so it's really helpful that TypeScript helps us catch it. But if object types are covariant in their members, how is it that TypeScript catches this?

That is:

- We expected the type `{baseURL: string, cacheSize?: number, tier?: 'prod' | 'dev'}`.
- We passed in the type `{baseURL: string, tierr: string}`.
- The type we passed in is a subtype of the type we expected, but somehow, TypeScript knew to report an error.

TypeScript was able to catch this due to its *excess property checking*, which works like this: when you try to assign a fresh object literal type `T` to another type `U`, and `T` has properties that aren't present in `U`, TypeScript reports an error.

A *fresh object literal type* is the type TypeScript infers from an object literal. If that object literal either uses a type assertion (see "Type Assertions") or is assigned to a variable, then the fresh object literal type is *widened* to a regular object type, and its freshness disappears.

This definition is dense, so let's walk through our example again, trying a few more variations on the theme this time:

```
type Options = {
  baseURL: string
  cacheSize?: number
  tier?: 'prod' | 'dev'
}

class API {
  constructor(private options: Options) {}
}

new API({
                                        ❶

  baseURL: 'https://api.mysite.com',
  tier: 'prod'
})
```

```
new API({
                                      ❷

  baseURL: 'https://api.mysite.com',
  badTier: 'prod'     // Error TS2345: Argument of type
})                    // string}' is not assignable to p

new API({
                                      ❸

  baseURL: 'https://api.mysite.com',
  badTier: 'prod'
} as Options)

let badOptions = {
                                      ❹

  baseURL: 'https://api.mysite.com',
  badTier: 'prod'
}
new API(badOptions)

let options: Options = {
                                      ❺

  baseURL: 'https://api.mysite.com',
  badTier: 'prod'     // Error TS2322: Type '{baseURL: s
}                     // is not assignable to type 'Opti
new API(options)
```

We instantiate `API` with a `baseURL` and one of our two optional properties, `tier`. This works as expected.

Here, we misspell `tier` as `badTier`. The options object we pass to `new API` is fresh (because its type is inferred, it isn't assigned to a variable, and we don't make a type assertion about its type), so TypeScript runs an excess property check on it, revealing the excess `badTier` property (which is defined in our options object but not on the `Options` type).

❸ We assert that our invalid options object is of type `Options`. TypeScript no longer considers it fresh, and bails out of excess property checking: no error. If you're not familiar with the `as T` syntax, read ahead to "Type Assertions".

❹ We assign our options object to a variable, `badOptions`. TypeScript no longer considers it to be fresh, and bails out of excess property

checking: no error.
When we explicitly type `options` as `Options`, the object we assign to `options` is fresh, so TypeScript performs excess property checking, catching our bug. Note that in this case the excess property check doesn't happen when we pass `options` to `new API`; rather, it happens when we try to assign our options object to the variable `options`.

Don't worry—you don't need to memorize these rules. They are TypeScript's internal heuristics for catching the most bugs possible in a practical way, so as not to be a burden on you, the programmer. Just keep them in mind when you're wondering how TypeScript knew to complain about that one bug that even Ivan, the battle-weathered gatekeeper of your company's codebase and master code reviewer, didn't notice.

Refinement

TypeScript performs flow-based type inference, which is a kind of symbolic execution where the typechecker uses control flow statements like `if`, `?`, `||`, and `switch`, as well as type queries like `typeof`, `instanceof`, and `in`, to *refine* types as it goes, just like a programmer reading through the code would.[1] It's an incredibly convenient feature for a typechecker to have, but is another one of those things that remarkably few languages support.[2]

Let's walk through an example. Say we've built an API for defining CSS rules in TypeScript, and a coworker wants to use it to set an HTML element's `width`. They pass in the width, which we then want to parse and validate.

We'll first implement a function to parse a CSS string into a value and a unit:

```
// We use a union of string literals to describe
// the possible values a CSS unit can have
type Unit = 'cm' | 'px' | '%'

// Enumerate the units
let units: Unit[] = ['cm', 'px', '%']

// Check each unit, and return null if there is no matc
function parseUnit(value: string): Unit | null {
  for (let i = 0; i < units.length; i++) {
    if (value.endsWith(units[i])) {
      return units[i]
    }
  }
  return null
}
```

We can then use `parseUnit` to parse a width value passed to us by a user. `width` might be a number (which we assume is in pixels), or a string with units attached, or it might be `null` or `undefined`.

We take advantage of type refinement a few times in this example:

```
  type Width = {
    unit: Unit,
    value: number
  }

  function parseWidth(width: number | string | null | unc
    // If width is null or undefined, return early
    if (width == null) {
                                        ❶

      return null
    }

    // If width is a number, default to pixels
    if (typeof width === 'number') {
                                        ❷

      return {unit: 'px', value: width}
    }

    // Try to parse a unit from width
    let unit = parseUnit(width)
    if (unit) {
                                        ❸

      return {unit, value: parseFloat(width)}
    }

    // Otherwise, return null
    return null
                                        ❹

  }
```

❶

TypeScript is smart enough to know that doing a loose equality check against `null` will return `true` for both `null` and `undefined` in JavaScript. It knows that if this check passes then we will return, and if we didn't return that means the check didn't pass, so from then on `width`'s type is `number | string` (it can't be `null` or `undefined` anymore). We say that the type was refined from `number | string | null | undefined` to `number | string`.

A `typeof` check queries a value at runtime to see what its type is. TypeScript takes advantage of `typeof` at compile time too: in the `if` branch where the check passes, TypeScript knows that `width` is a `number`; otherwise (since that branch `return`s) `width` must be a `string`—it's the only type left.

Because calling `parseUnit` might return `null`, we check if it did by testing whether its result is truthy.[3] TypeScript knows that if `unit` is truthy then it must be of type `Unit` in the `if` branch—otherwise, `unit` must be falsy, meaning it must be of type `null` (refined from `Unit | null`).

Finally, we return `null`. This can only happen if the user passed a `string` for `width`, but that string contained a unit that we don't support.

I've spelled out exactly what TypeScript was thinking for each of the type refinements it performed here, but I hope this was already intuitive and obvious for you, the programmer reading that code. TypeScript does a superb job of taking what's going through your mind as you read and write code, and crystallizing it in the form of typechecking and inference rules.

Discriminated union types

As we just learned, TypeScript has a deep understanding of how JavaScript works, and is able to follow along as you refine your types, just like you would when you trace through your program in your head.

For example, say we're building a custom event system for an application. We start by defining a couple of event types, along with a function to handle events that come in. Imagine that `UserTextEvent` models a keyboard event (e.g., the user typed something in a text `<input />`) and

UserMouseEvent models a mouse event (e.g., the user moved their mouse to the coordinates [100, 200]):

```
type UserTextEvent = {value: string}
type UserMouseEvent = {value: [number, number]}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
  if (typeof event.value === 'string') {
    event.value  // string
    // ...
    return
  }
  event.value    // [number, number]
}
```

Inside the if block, TypeScript knows that event.value has to be a string (because of the typeof check), which implies that after the if block event.value has to be a tuple of [number, number] (because of the return in the if block).

What happens if we make this a little more complicated? Let's add some more information to our event types, and see how TypeScript fares when we refine our types:

```
type UserTextEvent = {value: string, target: HTMLInputE
type UserMouseEvent = {value: [number, number], target:

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
  if (typeof event.value === 'string') {
    event.value  // string
    event.target // HTMLInputElement | HTMLElement (!!
    // ...
    return
  }
  event.value    // [number, number]
  event.target   // HTMLInputElement | HTMLElement (!!
}
```

While the refinement worked for `event.value`, it didn't carry over to `event.target`. Why? When `handle` takes a parameter of type `UserEvent`, that doesn't mean we have to pass a `UserTextEvent` or `UserMouseEvent`—in fact, we could pass an argument of type `UserMouseEvent | UserTextEvent`. And since members of a union might overlap, TypeScript needs a more reliable way to know when we're in one case of a union type versus another case.

The way to do this is to use a literal type to *tag* each case of your union type. A good tag is:

- On the same place in each case of your union type. That means the same object field if it's a union of object types, or the same index if it's a union of tuple types. In practice, tagged unions usually use object types.
- Typed as a literal type (a literal string, number, boolean, etc.). You can mix and match different types of literals, but it's good practice to stick to a single type; typically, that's a string literal type.
- Not generic. Tags should not take any generic type arguments.
- Mutually exclusive (i.e., unique within the union type).

With that in mind, let's update our event types again:

```
type UserTextEvent = {type: 'TextEvent', value: string,
type UserMouseEvent = {type: 'MouseEvent', value: [numb
                       target: HTMLElement}

type UserEvent = UserTextEvent | UserMouseEvent

function handle(event: UserEvent) {
  if (event.type === 'TextEvent') {
    event.value  // string
    event.target // HTMLInputElement
    // ...
    return
  }
  event.value    // [number, number]
  event.target   // HTMLElement
}
```

Now when we refine `event` based on the value of its tagged field (`event.type`), TypeScript knows that in the `if` branch `event` has to be a

`UserTextEvent` , and after the `if` branch it has to be a `UserMouseEvent` . Since the tag is unique per union type, TypeScript knows that the two are mutually exclusive.

Use tagged unions when writing a function that has to handle the different cases of a union type. For example, they're invaluable when working with Flux actions, Redux reducers, or React's `useReducer` .

Totality

> *A programmer puts two glasses on her bedside table before going to sleep: a full one, in case she gets thirsty, and an empty one, in case she doesn't.*
>
> —Anonymous

Totality, also called *exhaustiveness checking*, is what allows the typechecker to make sure you've covered all your cases. It comes to us from Haskell, OCaml, and other languages that are based around pattern matching.

TypeScript will check for totality in a variety of cases, and give you helpful warnings when you've missed a case. This is an incredibly helpful feature for preventing real bugs. For example:

```
type Weekday = 'Mon' | 'Tue'| 'Wed' | 'Thu' | 'Fri'
type Day = Weekday | 'Sat' | 'Sun'

function getNextDay(w: Weekday): Day {
  switch (w) {
    case 'Mon': return 'Tue'
  }
}
```

We clearly missed a few days (it's been a long week). TypeScript comes to the rescue:

```
Error TS2366: Function lacks ending return statement ar
return type does not include 'undefined'.
```

TSC FLAG: NOIMPLICITRETURNS

To ask TypeScript to check that all of your functions' code paths return a value (and throw the preceding warning if you missed a spot), enable the `noImplicitReturns` flag in your *tsconfig.json*. Whether you enable this flag or not is up to you: some people prefer a code style with fewer explicit `return`s, and some people are fine with a few extra `return`s in the name of better type safety and more bugs caught by the typechecker.

This error message is telling us that either we missed some cases and should cover them with a catchall `return` statement at the end that returns something like `'Sat'` (that'd be nice, huh), or we should adjust `getNextDay`'s return type to `Day | undefined`. After we add a `case` for each `Day`, the error goes away (try it!). Because we annotated `getNextDay`'s return type, and not all branches are guaranteed to return a value of that type, TypeScript warns us.

The implementation details in this example aren't important: no matter what kind of control structure you use— `switch`, `if`, `throw`, and so on— TypeScript will watch your back to make sure you have every case covered.

Here's another example:

```
function isBig(n: number) {
  if (n >= 100) {
    return true
  }
}
```

Maybe a client's continued voicemails about that missed deadline have you jittery, and you forgot to handle numbers under `100` in your business-critical `isBig` function. Again, never fear—TypeScript is watching out for you:

```
Error TS7030: Not all code paths return a value.
```

Or maybe the weekend gave you a chance to clear your mind, and you realized that you should rewrite that `getNextDay` example from earlier to be more efficient. Instead of using a `switch`, why not a constant-time lookup in an object?

```
let nextDay = {
  Mon: 'Tue'
}

nextDay.Mon // 'Tue'
```

With your Bichon Frise yapping away in the other room (something about the neighbor's dog?), you absentmindedly forgot to fill in the other days in your new `nextDay` object before you committed your code and moved on to other things.

While TypeScript will give you an error the next time you try to access `nextDay.Tue`, you could have been more proactive about it when declaring `nextDay` in the first place. There are two ways to do that, as you'll learn in "The Record Type" and "Mapped Types"; but before we get there, let's take a slight detour into type operators for object types.

Advanced Object Types

Objects are central to JavaScript, and TypeScript gives you a whole bunch of ways to express and manipulate them safely.

Type Operators for Object Types

Remember union (`|`) and intersection (`&`), the two type operators I introduced in "Union and intersection types"? It turns out they're not the only type operators TypeScript gives you! Let's run through a few more type operators that come in handy for working with shapes.

The keying-in operator

Say you have a complex nested type to model the GraphQL API response you got back from your social media API of choice:

```
type APIResponse = {
  user: {
    userId: string
    friendList: {
      count: number
      friends: {
        firstName: string
        lastName: string
      }[]
    }
  }
}
```

You might fetch that response from the API, then render it:

```
function getAPIResponse(): Promise<APIResponse> {
  // ...
}

function renderFriendList(friendList: unknown) {
  // ...
}

let response = await getAPIResponse()
renderFriendList(response.user.friendList)
```

What should the type of `friendList` be? (It's stubbed out as `unknown` for now.) You could type it out and reimplement your top-level `APIResponse` type in terms of it:

```
type FriendList = {
  count: number
  friends: {
    firstName: string
    lastName: string
  }[]
}

type APIResponse = {
  user: {
    userId: string
    friendList: FriendList
  }
}

function renderFriendList(friendList: FriendList) {
  // ...
}
```

But then you'd have to come up with names for each of your top-level types, which you don't always want (e.g., if you used a build tool to generate TypeScript types from your GraphQL schema). Instead, you can *key in* to your type:

```
type APIResponse = {
  user: {
    userId: string
    friendList: {
      count: number
      friends: {
        firstName: string
        lastName: string
      }[]
    }
  }
}

type FriendList = APIResponse['user']['friendList']

function renderFriendList(friendList: FriendList) {
  // ...
}
```

You can key in to any shape (object, class constructor, or class instance), and any array. For example, to get the type of an individual friend:

```
type Friend = FriendList['friends'][number]
```

`number` is a way to key in to an array type; for tuples, use `0`, `1`, or another number literal type to represent the index you want to key in to.

The syntax for keying in is intentionally similar to how you look up fields in regular JavaScript objects—just as you might look up a value in an object, so you can look up a type in a shape. Note that you have to use bracket notation, not dot notation, to look up property types when keying in.

The keyof operator

Use `keyof` to get all of an object's keys as a union of string literal types. Using the previous `APIResponse` example:

```
type ResponseKeys = keyof APIResponse // 'user'
type UserKeys = keyof APIResponse['user'] // 'userId' |
type FriendListKeys =
  keyof APIResponse['user']['friendList'] // 'count' |
```

Combining the keying-in and `keyof` operators, you can implement a typesafe getter function that looks up the value at the given key in an object:

```
function get<
                         ❶

  O extends object,
  K extends keyof O
                         ❷

>(
  o: O,
  k: K
): O[K] {
                         ❸

  return o[k]
}
```

❶ `get` is a function that takes an object `o` and a key `k`.

❷ `keyof O` is a union of string literal types, representing all of `o`'s keys. The generic type `K` extends—and is a subtype of—that union. For example, if `o` has the type `{a: number, b: string, c: boolean}`, then `keyof o` is the type `'a' | 'b' | 'c'`, and `K` (which extends `keyof o`) could be the type `'a'`, `'b'`, `'a' | 'c'`, or any other subtype of `keyof o`.

❸ `O[K]` is the type you get when you look up `K` in `O`. Continuing the example from

❷

, if `K` is `'a'`, then we know at compile time that `get` returns a `number`. Or, if `K` is `'b' | 'c'`, then we know `get` returns `string | boolean`.

What's cool about these type operators is how precisely and safely they let you describe shape types:

```
type ActivityLog = {
  lastEvent: Date
  events: {
    id: string
    timestamp: Date
    type: 'Read' | 'Write'
  }[]
}

let activityLog: ActivityLog = // ...
let lastEvent = get(activityLog, 'lastEvent') // Date
```

TypeScript goes to work for you, verifying *at compile time* that the type of `lastEvent` is `Date`. Of course, you could extend this in order to key in to an object more deeply too. Let’s overload `get` to accept up to three keys:

```
type Get = {
                                    ❶

  <
    O extends object,
    K1 extends keyof O
  >(o: O, k1: K1): O[K1]
                                    ❷

  <
    O extends object,
    K1 extends keyof O,
    K2 extends keyof O[K1]
                                    ❸

  >(o: O, k1: K1, k2: K2): O[K1][K2]
                                    ❹

  <
    O extends object,
    K1 extends keyof O,
    K2 extends keyof O[K1],
    K3 extends keyof O[K1][K2]
  >(o: O, k1: K1, k2: K2, k3: K3): O[K1][K2][K3]
                                    ❺

}

let get: Get = (object: any, ...keys: string[]) => {
  let result = object
```

```
    keys.forEach(k => result = result[k])
    return result
  }

  get(activityLog, 'events', 0, 'type') // 'Read' | 'Writ


  get(activityLog, 'bad') // Error TS2345: Argument of ty
                          // is not assignable to paramet
                          // '"lastEvent" | "events"'.
```

❶ We declare an overloaded function signature for `get` with three cases for when we call `get` with one key, two keys, and three keys.

❷ This one-key case is the same as the last example: `O` is a subtype of `object`, `K1` is a subtype of that object's keys, and the return type is whatever specific type you get when you key in to `O` with `K1`.

❸ The two-key case is like the one-key case, but we declare one more generic type, `K2`, to model the possible keys on the nested object that results from keying into `O` with `K1`.

❹ We build on ❷ by keying in twice—we first get the type of `O[K1]`, then get the type of `[K2]` on the result.

❺ For this example we handle up to three nested keys; if you're writing a real-world library, you'll probably want to handle a few more cases than that.

Cool, huh? If you have a minute, show this example to your Java friends, and be sure to gloat as you walk them through it.

TSC FLAG: KEYOFSTRINGSONLY

In JavaScript, objects and arrays can have both string and symbol keys. And by convention, we usually use number keys for arrays, which are coerced to strings at runtime.

Because of this, `keyof` in TypeScript returns a value of type `number | string | symbol` by default (though if you call it on a more specific shape, TypeScript can infer a more specific subtype of that union).

This behavior is correct, but can make working with `keyof` wordy, as you may have to prove to TypeScript that the particular key you're manipulating is a `string`, and not a `number` or a `symbol`.

To opt into TypeScript's legacy behavior—where keys must be strings—enable the `keyofStringsOnly` *tsconfig.json* flag.

The Record Type

TypeScript's built-in `Record` type is a way to describe an object as a map from something to something.

Recall from the `Weekday` example in "Totality" that there are two ways to enforce that an object defines a specific set of keys. `Record` types are the first.

Let's use `Record` to build a map from each day of the week to the next day of the week. With `Record`, you can put some constraints on the keys and values in `nextDay`:

```
type Weekday = 'Mon' | 'Tue'| 'Wed' | 'Thu' | 'Fri'
type Day = Weekday | 'Sat' | 'Sun'

let nextDay: Record<Weekday, Day> = {
  Mon: 'Tue'
}
```

Now, you get a nice, helpful error message right away:

```
Error TS2739: Type '{Mon: "Tue"}' is missing the follow
```

```
from type 'Record<Weekday, Day>': Tue, Wed, Thu, Fri.
```

Adding the missing `Weekday` s to your object, of course, makes the error go away.

`Record` gives you one extra degree of freedom compared to regular object index signatures: with a regular index signature you can constrain the types of an object's values, but the key can only be a regular `string` , `number` , or `symbol` ; with `Record` , you can also constrain the types of an object's keys to subtypes of `string` and `number` .

Mapped Types

TypeScript gives us a second, more powerful way to declare a safer `nextDay` type: mapped types. Let's use mapped types to say that `nextDay` is an object with a key for each `Weekday` , whose value is a `Day` :

```
let nextDay: {[K in Weekday]: Day} = {
  Mon: 'Tue'
}
```

This is another way to get a helpful hint for how to fix what you missed:

```
Error TS2739: Type '{Mon: "Tue"}' is missing the follow
from type '{Mon: Weekday; Tue: Weekday; Wed: Weekday; T
Fri: Weekday}': Tue, Wed, Thu, Fri.
```

Mapped types are a language feature unique to TypeScript. Like literal types, they're a utility feature that just makes sense for the challenge that is statically typing JavaScript.

As you saw, mapped types have their own special syntax. And like index signatures, you can have at most one mapped type per object:

```
type MyMappedType = {
  [Key in UnionType]: ValueType
}
```

As the name implies, it’s a way to map over an object’s key and value types. In fact, TypeScript uses mapped types to implement its built-in `Record` type we used earlier:

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

Mapped types give you more power than a mere `Record` because in addition to letting you give types to an object’s keys and values, when you combine them with keyed-in types, they let you put constraints on which value type corresponds to which key name.

Let’s quickly run through some of the things you can do with mapped types.

```
type Account = {
  id: number
  isEmployee: boolean
  notes: string[]
}

// Make all fields optional
type OptionalAccount = {
  [K in keyof Account]?: Account[K]
                          ❶

}

// Make all fields nullable
type NullableAccount = {
  [K in keyof Account]: Account[K] | null
                          ❷

}

// Make all fields read-only
type ReadonlyAccount = {
  readonly [K in keyof Account]: Account[K]
                          ❸

}

// Make all fields writable again (equivalent to Accour
type Account2 = {
  -readonly [K in keyof ReadonlyAccount]: Account[K]
                      ❹

}

// Make all fields required again (equivalent to Accour
type Account3 = {
  [K in keyof OptionalAccount]-?: Account[K]
                      ❺

}
```

We create a new object type `OptionalAccount` by mapping over `Account`, marking each field as optional along the way.

We create a new object type `NullableAccount` by mapping over `Account`, adding `null` as a possible value for each field along the way.

We create a new object type `ReadonlyAccount` by taking `Account` and making each of its fields read-only (that is, readable but not writable).

We can mark fields as optional (`?`) or `readonly`, and we can also unmark them. With the minus (`–`) operator—a special type operator only available with mapped types—we can undo `?` and `readonly`, making fields required and writable again, respectively. Here we create a new object type `Account2`, equivalent to our `Account` type, by mapping over `ReadonlyAccount` and removing the `readonly` modifier with the minus (`–`) operator.

We create a new object type `Account3`, equivalent to our original `Account` type, by mapping over `OptionalAccount` and removing the optional (`?`) operator with the minus (`–`) operator.

NOTE

Minus (`–`) has a corresponding plus (`+`) type operator. You will probably never use this operator directly, because it's implied: within a mapped type, `readonly` is equivalent to `+readonly`, and `?` is equivalent to `+?`. `+` is just there for completeness.

Built-in mapped types

The mapped types we derived in the last section are so useful that TypeScript ships with many of them built in:

`Record<Keys, Values>`
: An object with keys of type `Keys` and values of type `Values`

`Partial<Object>`
: Marks every field in `Object` as optional

`Required<Object>`
: Marks every field in `Object` as nonoptional

`Readonly<Object>`
: Marks every field in `Object` as read-only

`Pick<Object, Keys>`
: Returns a subtype of `Object`, with just the given `Keys`

Companion Object Pattern

The companion object pattern comes to us from Scala, and is a way to pair together objects and classes that share the same name. In TypeScript, there's a similar pattern that's similarly useful—we'll also call it the companion object pattern—that we can use to pair together a type and an object.

It looks like this:

```
type Currency = {
  unit: 'EUR' | 'GBP' | 'JPY' | 'USD'
  value: number
}

let Currency = {
  DEFAULT: 'USD',
  from(value: number, unit = Currency.DEFAULT): Curren
    return {unit, value}
  }
}
```

Remember that in TypeScript, types and values live in separate namespaces; you'll read a little more about this in "Declaration Merging". That means in the same scope, you can have the same name (in this example, `Currency`) bound to both a type and a value. With the companion object pattern, we exploit this separate namespacing to declare a name twice: first as a type, then as a value.

This pattern has a few nice properties. It lets you group type and value information that's semantically part of a single name (like `Currency`) together. It also lets consumers import both at once:

```
import {Currency} from './Currency'

let amountDue: Currency = {
                                    ❶

  unit: 'JPY',
  value: 83733.10
}

let otherAmountDue = Currency.from(330, 'EUR')
                                    ❷
```

❶ Using `Currency` as a type

❷ Using `Currency` as a value

Use the companion object pattern when a type and an object are semantically related, with the object providing utility methods that operate on the type.

Advanced Function Types

Let's take a look at a few more advanced techniques that are often used with function types.

Improving Type Inference for Tuples

When you declare a tuple in TypeScript, TypeScript will be lenient about inferring that tuple's type. It will infer the most general possible type based on what you gave it, ignoring the length of your tuple and which position holds which type:

```
let a = [1, true] // (number | boolean)[]
```

But sometimes you want inference that's stricter, that would treat `a` as a fixed-length tuple and not as an array. You could, of course, use a type assertion to cast your tuple to a tuple type (more on this in "Type Assertions"). Or, you could use an `as const` assertion ("The const type") to infer the tuple's type as narrowly as possible, marking it as read-only.

What if you want to type your tuple as a tuple, but avoid a type assertion, and avoid the narrow inference and read-only modifier that `as const` gives you? To do that, you can take advantage of the way TypeScript infers types for rest parameters (jump back to "Using bounded polymorphism to model arity" for more about that):

```
function tuple<
                                 ❶
  T extends unknown[]
                                 ❷
>(
  ...ts: T
                                 ❸
): T {
                                 ❹
  return ts
                                 ❺
}

let a = tuple(1, true) // [number, boolean]
```

❶ We declare a `tuple` function that we'll use to construct tuple types (instead of using the built-in `[]` syntax).

❷ We declare a single type parameter `T` that's a subtype of `unknown[]` (meaning `T` is an array of any kind of type).

❸ `tuple` takes a variable number of parameters, `ts`. Because `T` describes a rest parameter, TypeScript will infer a tuple type for it.

❹ `tuple` returns a value of the same tuple type that it inferred `ts` as.

❺ Our function returns the same argument that we passed it. The magic is all in the types.

Take advantage of this technique in order to avoid type assertions when your code uses lots of tuple types.

User-Defined Type Guards

For some kinds of `boolean` -returning functions, simply saying that your function returns a `boolean` may not be enough. For example, let's write a function that tells you if you passed it a `string` or not:

```
function isString(a: unknown): boolean {
  return typeof a === 'string'
}

isString('a') // evaluates to true
isString([7]) // evaluates to false
```

So far so good. What happens if you try to use `isString` in some real-world code?

```
function parseInput(input: string | number) {
  let formattedInput: string
  if (isString(input)) {
    formattedInput = input.toUpperCase() // Error TS23
  }                                      // does not e
}
```

What gives? If `typeof` works for regular type refinement (see "Refinement"), why doesn't it work here?

The thing about type refinement is it's only powerful enough to refine the type of a variable in the scope you're in. As soon as you leave that scope, the refinement doesn't carry over to whatever new scope you're in. In our `isString` implementation, we refined the input parameter's type to `string` using `typeof`, but because type refinement doesn't carry over to new scopes, it got lost—all TypeScript knows is that `isString` returned a `boolean`.

What we can do is tell the typechecker that not only does `isString` return a `boolean`, but whenever that `boolean` is `true`, the argument we passed to `isString` is a `string`. To do that, we use something called a *user-defined type guard*:

```
function isString(a: unknown): a is string {
  return typeof a === 'string'
}
```

Type guards are a built-in TypeScript feature, and are what lets you refine types with `typeof` and `instanceof`. But sometimes, you need the ability to declare type guards yourself—that's what the `is` operator is for. When you have a function that refines its parameters' types and returns a `boolean`, you can use a user-defined type guard to make sure that refinement is flowed whenever you use that function.

User-defined type guards are limited to a single parameter, but they aren't limited to simple types:

```
type LegacyDialog = // ...
type Dialog = // ...

function isLegacyDialog(
  dialog: LegacyDialog | Dialog
): dialog is LegacyDialog {
  // ...
}
```

You won't use user-defined type guards often, but when you do, they're awesome for writing clean, reusable code. Without them, you'd have to inline all your `typeof` and `instanceof` type guards instead of building functions like `isLegacyDialog` and `isString` to perform those same checks in a better-encapsulated, more readable way.

Conditional Types

Conditional types might be the single most unique feature in all of TypeScript. At a high level, conditional types let you say, "Declare a type `T` that depends on types `U` and `V`; if `U <: V`, then assign `T` to `A`, and otherwise, assign `T` to `B`."

In code it might look like this:

```
type IsString<T> = T extends string
                                   ❶
    ? true
                                   ❷
    : false
                                   ❸

type A = IsString<string> // true
type B = IsString<number> // false
```

Let’s break that down line by line.

❶ We declare a new conditional type `IsString` that takes a generic type `T`. The “condition” part of this conditional type is `T extends string`; that is, “Is `T` a subtype of `string`?”

❷ If `T` is a subtype of `string`, we resolve to the type `true`.

❸ Otherwise, we resolve to the type `false`.

Note how the syntax looks just like a regular value-level ternary expression, but at the type level. And like regular ternary expressions, you can nest them too.

Conditional types aren’t limited to type aliases. You can use them almost anywhere you can use a type: in type aliases, interfaces, classes, parameter types, and generic defaults in functions and methods.

Distributive Conditionals

While you can express simple conditions like the examples we just looked at in a variety of ways in TypeScript—with conditional types, overloaded function signatures, and mapped types—conditional types let you do more. The reason for this is that they follow the *distributive law* (remember, from algebra class?). That means if you have a conditional type, then the expressions on the right are equivalent to those on the left in Table 6-1.

Table 6-1. Distributing conditional types

This…	Is equivalent to
`string extends T ? A : B`	`string extends T ? A : B`
`(string | number) extends T ? A : B`	`(string extends T ? A : B) | (number extends T ? A : B)`
`(string | number | boolean) extends T ? A : B`	`(string extends T ? A : B) | (number extends T ? A : B) | (boolean extends T ? A : B)`

I know, I know, you didn't shell out for this book to learn about math—you're here for the types. So let's get more concrete. Let's say we have a function that takes some variable of type `T`, and lifts it to an array of type `T[]`. What happens if we pass in a union type for `T`?

```
type ToArray<T> = T[]
type A = ToArray<number>          // number[]
type B = ToArray<number | string> // (number | string)[
```

Pretty straightforward. Now what happens if we add a conditional type? (Note that the conditional doesn't actually do anything here because both its branches resolve to the same type `T[]`; it's just here to tell TypeScript to *distribute* `T` over the tuple type.) Take a look:

```
type ToArray2<T> = T extends unknown ? T[] : T[]
type A = ToArray2<number> // number[]
type B = ToArray2<number | string> // number[] | strin
```

Did you catch that? When you use a conditional type, TypeScript will distribute union types over the conditional's branches. It's like taking the conditional type and mapping (er, *distributing*) it over each element in the union.

Why does any of this matter? Well, it lets you safely express a bunch of common operations.

For example, TypeScript comes with `&` for computing what two types have in common and `|` for taking a union of two types. Let's build `Without<T, U>`, which computes the types that are in `T` but not in `U`.

```
type Without<T, U> = T extends U ? never : T
```

You use `Without` like so:

```
type A = Without<
  boolean | number | string,
  boolean
> // number | string
```

Let's walk through how TypeScript computes this type:

1. Start with the inputs:

   ```
   type A = Without<boolean | number | string, boolean>
   ```

2. Distribute the condition over the union:

   ```
   type A = Without<boolean, boolean>
          | Without<number, boolean>
          | Without<string, boolean>
   ```

3. Substitute in `Without`'s definition and apply `T` and `U`:

   ```
   type A = (boolean extends boolean ? never : boolean)
          | (number extends boolean ? never : number)
          | (string extends boolean ? never : string)
   ```

4. Evaluate the conditions:

   ```
   type A = never
          | number
          | string
   ```

5. Simplify:

```
type A = number | string
```

If it wasn't for the distributive property of conditional types, we would have ended up with `never` (if you're not sure why, walk through what would happen for yourself!).

The infer Keyword

The final feature of conditional types is the ability to declare generic types as part of a condition. As a refresher, so far we've seen just one way to declare generic type parameters: using angle brackets (`<T>`). Conditional types have their own syntax for declaring generic types inline: the `infer` keyword.

Let's declare a conditional type `ElementType` , which gets the type of an array's elements:

```
type ElementType<T> = T extends unknown[] ? T[number] :
type A = ElementType<number[]> // number
```

Now, let's rewrite it using `infer` :

```
type ElementType2<T> = T extends (infer U)[] ? U : T
type B = ElementType2<number[]> // number
```

In this simple example `ElementType` is equivalent to `ElementType2` . Notice how the `infer` clause declares a new type variable, `U` —TypeScript will infer the type of `U` from context, based on what `T` you passed to `ElementType2` .

Also notice why we declared `U` inline instead of declaring it up front, alongside `T` . What would have happened if we did declare it up front?

```
type ElementUgly<T, U> = T extends U[] ? U : T
type C = ElementUgly<number[]> // Error TS2314: Generic
```

```
                                      // requires 2 type argum
```

Uh-oh. Because `ElementUgly` defines two generic types, `T` and `U`, we have to pass both of them in when instantiating `ElementUgly`. But if we do that, that defeats the point of having an `ElementUgly` type in the first place; it puts the burden of computing `U` on the caller, when we wanted `ElementUgly` to compute the type itself.

Honestly, this was a bit of a silly example because we already have the keying-in operator (`[]`) to look up the type of an array's elements. What about a more complicated example?

```
type SecondArg<F> = F extends (a: any, b: infer B) => a

// Get the type of Array.slice
type F = typeof Array['prototype']['slice']

type A = SecondArg<F> // number | undefined
```

So, `[].slice`'s second argument is a `number | undefined`. And we know this at compile time—try doing *that* in Java.

Built-in Conditional Types

Conditional types let you express some really powerful operations at the type level. That's why TypeScript ships with a few globally available conditional types out of the box:

`Exclude<T, U>`
: Like our `Without` type from before, computes those types in `T` that are not in `U`:

```
type A = number | string
type B = string
type C = Exclude<A, B>  // number
```

`Extract<T, U>`
: Computes the types in `T` that you can assign to `U`:

```
type A = number | string
type B = string
type C = Extract<A, B>  // string
```

`NonNullable<T>`

Computes a version of `T` that excludes `null` and `undefined`:

```
type A = {a?: number | null}
type B = NonNullable<A['a']>  // number
```

`ReturnType<F>`

Computes a function's return type (note that this doesn't work as you'd expect for generic and overloaded functions):

```
type F = (a: number) => string
type R = ReturnType<F>  // string
```

`InstanceType<C>`

Computes the instance type of a class constructor:

```
type A = {new(): B}
type B = {b: number}
type I = InstanceType<A>  // {b: number}
```

Escape Hatches

Sometimes you don't have time to type something perfectly, and you just want TypeScript to trust that what you're doing is safe. Maybe a type declaration for a third party module you're using is wrong and you want to test your code before contributing the fix back to DefinitelyTyped,[4] or maybe you're getting data from an API and you haven't regenerated type declarations with Apollo yet.

Luckily, TypeScript knows that we're only human, and gives us a few escape hatches for when we just want to do something and don't have time to prove to TypeScript that it's safe.

NOTE

In case it's not obvious, you should use the following TypeScript features as little as possible. If you find yourself relying on them, you might be doing something wrong.

Type Assertions

If you have a type `B` and `A <: B <: C`, then you can assert to the typechecker that `B` is actually an `A` or a `C`. Notably, you can only assert that a type is a supertype or a subtype of itself—you can't, for example, assert that a `number` is a `string`, because those types aren't related.

TypeScript gives us two syntaxes for type assertions:

```
function formatInput(input: string) {
  // ...
}

function getUserInput(): string | number {
  // ...
}

let input = getUserInput()

// Assert that input is a string
formatInput(input as string)
❶


// This is equivalent to
formatInput(<string>input)
❷
```

❶ We use a type assertion (`as`) to tell TypeScript that `input` is a `string`, not a `string | number` as the types would have us believe. You might do this, for example, if you want to quickly test out your `formatInput` function and you know for sure that `getUserInput` returns a `string` for your test.

❷ The legacy syntax for type assertions uses angle brackets. The two syntaxes are functionally equivalent.

NOTE

Prefer `as` syntax for type assertions over angle bracket (`<>`) syntax. The former is unambiguous, but the latter can clash with TSX syntax (see "TSX = JSX + TypeScript"). Use TSLint's `no-angle-bracket-type-assertion` rule to automatically enforce this for your codebase.

Sometimes, two types might not be sufficiently related, so you can't assert that one is the other. To get around this, simply assert as `any` (remember from "Assignability" that `any` is assignable to anything), then spend a few minutes in the corner thinking about what you've done:

```
function addToList(list: string[], item: string) {
  // ...
}

addToList('this is really,' as any, 'really unsafe')
```

Clearly, type assertions are unsafe, and you should avoid using them when possible.

Nonnull Assertions

For the special case of nullable types—that is, a type that's `T | null` or `T | null | undefined`—TypeScript has special syntax for asserting that a value of that type is a `T`, and not `null` or `undefined`. This comes up in a few places.

For example, say we've written a framework for showing and hiding dialogs in a web app. Each dialog gets a unique ID, which we use to get a reference to the dialog's DOM node. Once a dialog is removed from the DOM, we delete its ID, indicating that it's no longer live in the DOM:

```
type Dialog = {
  id?: string
}

function closeDialog(dialog: Dialog) {
  if (!dialog.id) {
```
❶

```
      return
    }
    setTimeout(() =>
                                ❷

      removeFromDOM(
        dialog,
        document.getElementById(dialog.id) // Error TS234
                                           // 'string | u
                                           // to paramete
                                ❸

      )
    )
}

function removeFromDOM(dialog: Dialog, element: Element
    element.parentNode.removeChild(element) // Error TS25
                                            //'null'.
                                ❹

    delete dialog.id
}
```

❶ If the dialog is already deleted (so it has no `id`), we return early.

❷ We remove the dialog from the DOM on the next turn of the event loop, so that any other code that depends on `dialog` has a chance to finish running.

❸ Because we're inside the arrow function, we're now in a new scope. TypeScript doesn't know if some code mutated `dialog` between

❶

and

❸

, so it invalidates the refinement we made in

❶

. On top of that, while we know that if `dialog.id` is defined then an element with that ID definitely exists in the DOM (because we designed our framework that way), all TypeScript knows is that calling `document. getElementById` returns an `HTMLElement | null`. We know it'll always be a nonnullable `HTMLElement`, but TypeScript doesn't know that—it only knows about the types we gave it.

❹ Similarly, while we know that the dialog is definitely in the DOM and it definitely has a parent DOM node, all TypeScript knows is that the type of `element.parentNode` is `Node | null`.

One way to fix this is to add a bunch of `if (_ === null)` checks everywhere. While that's the right way to do it if you're unsure if something is `null` or not, TypeScript comes with special syntax for when you're sure it's not `null | undefined`:

```
type Dialog = {
  id?: string
}

function closeDialog(dialog: Dialog) {
  if (!dialog.id) {
    return
  }
  setTimeout(() =>
    removeFromDOM(
      dialog,
      document.getElementById(dialog.id!)!
    )
  )
}

function removeFromDOM(dialog: Dialog, element: Element
  element.parentNode!.removeChild(element)
  delete dialog.id
}
```

Notice the sprinkling of non `null` assertion operators (`!`) that tell TypeScript that we're sure `dialog.id`, the result of our `document.getElementById` call, and `element.parentNode` are defined. When a non `null` assertion follows a type that might be `null` or `undefined`, TypeScript will assume that the type is defined: `T | null | undefined` becomes a `T`, `number | string | null` becomes `number | string`, and so on.

When you find yourself using non `null` assertions a lot, it's often a sign that you should refactor your code. For example, we could get rid of an assertion by splitting `Dialog` into a union of two types:

```
type VisibleDialog = {id: string}
type DestroyedDialog = {}
type Dialog = VisibleDialog | DestroyedDialog
```

We can then update `closeDialog` to take advantage of the union:

```
function closeDialog(dialog: Dialog) {
  if (!('id' in dialog)) {
    return
  }
  setTimeout(() =>
    removeFromDOM(
      dialog,
      document.getElementById(dialog.id)!
    )
  )
}

function removeFromDOM(dialog: VisibleDialog, element:
  element.parentNode!.removeChild(element)
  delete dialog.id
}
```

After we check that `dialog` has an `id` property defined—implying that it's a `Visible Dialog` —even inside the arrow function TypeScript knows that the reference to `dialog` hasn't changed: the `dialog` inside the arrow function is the same `dialog` outside the function, so the refinement carries over instead of being invalidated like it was in the last example.

Definite Assignment Assertions

TypeScript has special syntax for the special case of non `null` assertions for definite assignment checks (as a reminder, a definite assignment check is TypeScript's way of making sure that by the time you use a variable, that variable has been assigned a value). For example:

```
let userId: string
```

```
userId.toUpperCase() // Error TS2454: Variable 'userId'
                     // before being assigned.
```

Clearly, TypeScript just did us a great service by catching this error. We declared the variable `userId`, but forgot to assign a value to it before we tried to convert it to uppercase. This would have been a runtime error if TypeScript hadn't noticed it!

But, what if our code looks more like this?

```
let userId: string
fetchUser()

userId.toUpperCase() // Error TS2454: Variable 'userId'
                     // before being assigned.

function fetchUser() {
  userId = globalCache.get('userId')
}
```

We happen to have the world's greatest cache, and when we query this cache we get a cache hit 100% of of the time. So after the call to `fetchUser`, `userId` is guaranteed to be defined. But TypeScript isn't able to statically detect that, so it still throws the same error as before. We can use a definite assignment assertion to tell TypeScript that `userId` will definitely be assigned by the time we read it (notice the exclamation mark):

```
let userId!: string
fetchUser()

userId.toUpperCase() // OK

function fetchUser() {
  userId = globalCache.get('userId')
}
```

As with type assertions and non `null` assertions, if you find yourself using definite assignment assertions often, you might be doing something wrong.

Simulating Nominal Types

By this point in the book, if I were to shake you awake at three in the morning and yell "IS TYPESCRIPT'S TYPE SYSTEM STRUCTURAL OR NOMINAL?!" you'd yell back "OF COURSE IT'S STRUCTURAL! NOW GET OUT OF MY HOUSE OR I'LL CALL THE POLICE!" That would be a fair response to me breaking in for early morning type system questions.

Laws aside, the reality is that sometimes nominal types really are useful. For example, let's say you have a few `ID` types in your application, representing unique ways of addressing the different types of objects in your system:

```
type CompanyID = string
type OrderID = string
type UserID = string
type ID = CompanyID | OrderID | UserID
```

A value of type `UserID` might be a simple hash that looks like `"d21b1dbf"`. So while you might alias it as `UserID`, under the hood it's of course just a regular `string`. A function that takes a `UserID` might look like this:

```
function queryForUser(id: UserID) {
  // ...
}
```

This is great documentation, and it helps other engineers on your team know for sure which type of `ID` they should pass in. But since `UserID` is just an alias for `string`, this approach does little to prevent bugs. An engineer might accidentally pass in the wrong type of `ID`, and the types system will be none the wiser!

```
let id: CompanyID = 'b4843361'
queryForUser(id) // OK (!!!)
```

This is where *nominal types* come in handy.[5] While TypeScript doesn't support nominal types out of the box, we can simulate them with a technique called *type branding*. Type branding takes a little work to set up, and using it in TypeScript is not as smooth an experience as it is in languages that have built-in support for nominal type aliases. That said, branded types can make your program significantly safer.

NOTE

Depending on your application and the size of your engineering team (the larger your team, the more likely this technique will come in handy for preventing mistakes), you may not need to do this.

Start by creating a synthetic *type brand* for each of your nominal types:

```
type CompanyID = string & {readonly brand: unique symbo
type OrderID = string & {readonly brand: unique symbol}
type UserID = string & {readonly brand: unique symbol}
type ID = CompanyID | OrderID | UserID
```

An intersection of `string` and `{readonly brand: unique symbol}` is, of course, gibberish. I chose it because it's impossible to naturally construct that type, and the only way to create a value of that type is with an assertion. That's the crucial property of branded types: they make it hard to accidentally use a wrong type in their place. I used `unique symbol` as the "brand" because it's one of two truly nominal kinds of types in TypeScript (the other is `enum`); I took an intersection of that brand with `string` so that we can assert that a given `string` is a given branded type.

We now need a way to create values of type `CompanyID`, `OrderID`, and `UserID`. To do that, we'll use the companion object pattern (introduced in "Companion Object Pattern"). We'll make a constructor for each branded type, using a type assertion to construct a value of each of our gibberish types:

```
function CompanyID(id: string) {
  return id as CompanyID
}

function OrderID(id: string) {
  return id as OrderID
}
```

```
function UserID(id: string) {
  return id as UserID
}
```

Finally, let's see what it feels like to use these types:

```
function queryForUser(id: UserID) {
  // ...
}

let companyId = CompanyID('8a6076cf')
let orderId = OrderID('9994acc1')
let userId = UserID('d21b1dbf')

queryForUser(userId)    // OK
queryForUser(companyId) // Error TS2345: Argument of ty
                        // assignable to parameter of t
```

What's nice about this approach is how little runtime overhead it has: just one function call per `ID` construction, which will probably be inlined by your JavaScript VM anyway. At runtime, each `ID` is simply a `string`—the brand is purely a compile-time construct.

Again, for most applications this approach is overkill. But for large applications, and when working with easily confused types like different kinds of IDs, branded types can be a killer safety feature.

Safely Extending the Prototype

When building JavaScript applications, tradition holds that it's unsafe to extend prototypes for built-in types. This rule of thumb goes back to before the days of jQuery, when wise JavaScript mages built libraries like MooTools that extended and overwrote built-in prototype methods directly. But when too many mages augmented prototypes at once, conflicts arose. And without static type systems, you'd only find out about these conflicts from angry users at runtime.

If you're not coming from JavaScript, you may be surprised to learn that in JavaScript, you can modify any built-in method (like `[].push`, `'abc'.toUpperCase`, or `Object.assign`) at runtime. Because it's such a dynamic language, JavaScript gives you direct access to prototypes for every built-in object— `Array.prototype` , `Function.prototype` , `Object.prototype` , and so on.

While back in the day extending these prototypes was unsafe, if your code is covered by a static type system like TypeScript, then you can now do it safely.[6]

For example, we'll add a `zip` method to the `Array` prototype. It takes two things to safely extend the prototype. First, in a *.ts* file (say, *zip.ts*), we extend the type of `Array` 's prototype; then, we augment the prototype with our new `zip` method:

```
// Tell TypeScript about .zip
interface Array<T> {                      ❶
  zip<U>(list: U[]): [T, U][]
}

// Implement .zip
Array.prototype.zip = function<T, U>(
  this: T[],                              ❷
  list: U[]
): [T, U][] {
  return this.map((v, k) =>
    tuple(v, list[k])                     ❸
  )
}
```

❶ We start by telling TypeScript that we're adding `zip` to `Array` . We take advantage of interface merging ("Declaration Merging") to augment the global `Array<T>` interface, adding our own `zip` method to the already globally defined interface.
Since our file doesn't have any explicit imports or exports—meaning it's in script mode, as described in "Module Mode Versus Script Mode"—we were able to augment the global `Array` interface directly by declaring an interface with the exact same name as the existing `Array<T>` interface, and letting TypeScript take care of merging the two for us. If our file were in module mode (which might

be the case if, for example, we needed to `import` something for our `zip` implementation), we'd have to wrap our global extension in a `declare global` type declaration (see "Type Declarations"):

```
declare global {
  interface Array<T> {
    zip<U>(list: U[]): [T, U][]
  }
}
```

`global` is a special namespace containing all the globally defined values (anything that you can use in a module-mode file without `import`ing it first; see Chapter 10) that lets you augment names in the global scope from a [illegible] module m[illegible]

We then implement the `zip` method on `Array`'s prototype. We use a `this` type so that TypeScript correctly infers the `T` type of the array we're calling `.zip` on. ❷

Because TypeScript infers the mapping function's return type as `(T | U)[]` (TypeScript isn't smart enough to realize that it's in fact always a tuple with `T` in the zeroth index and `U` in the first), we use our `tuple` utility (from "Improving Type Inference for Tuples") to create a tuple type without resorting to a type assertion. ❸

Notice that when we declare `interface Array<T>` we augment the global `Array` namespace for our whole TypeScript project—meaning even if we don't import *zip.ts* from our file, TypeScript will think that `[].zip` is available. But in order to augment `Array.prototype`, we have to be sure that whatever file uses `zip` loads *zip.ts* first, in order to install the `zip` method on `Array.prototype`. How do we make sure that any file that uses `zip` loads *zip.ts* first?

Easy: we update our *tsconfig.json* to explicitly exclude *zip.ts* from our project, so that consumers have to explicitly `import` it first:

```
{
  *exclude*: [
    "./zip.ts"
  ]
}
```

Now we can use `zip` as we please, with total safety:

```
import './zip'

[1, 2, 3]
  .map(n => n * 2)         // number[]
  .zip(['a', 'b', 'c'])    // [number, string][]
```

Running this gives us the result of first mapping, then zipping the array:

```
[
  [2, 'a'],
  [4, 'b'],
  [6, 'c']
]
```

Summary

In this chapter we covered the most advanced features of TypeScript's type system: from the ins and outs of variance to flow-based type inference, refinement, type widening, totality, and mapped and conditional types. We then derived a few advanced patterns for working with types: type branding to simulate nominal types, taking advantage of the distributive property of conditional types to operate on types at the type level, and safely extending prototypes.

If you didn't understand or don't remember everything, that's OK—come back to this chapter later, and use it as a reference when you're struggling with how to express something more safely.

Exercises

1. For each of the following pairs of types, decide if the first type is assignable to the second type, and why or why not. Think about these in terms of subtyping and variance, and refer to the rules at the start of the chapter if you're unsure (if you're still unsure, just type it into your code editor to check!):
   1. `1` and `number`

2. `number` and `1`
3. `string` and `number | string`
4. `boolean` and `number`
5. `number[]` and `(number | string)[]`
6. `(number | string)[]` and `number[]`
7. `{a: true}` and `{a: boolean}`
8. `{a: {b: [string]}}` and `{a: {b: [number | string]}}`
9. `(a: number) => string` and `(b: number) => string`
10. `(a: number) => string` and `(a: string) => string`
11. `(a: number | string) => string` and `(a: string) => string`
12. `E.X` (defined in an enum `enum E {X = 'X'}`) and `F.X` (defined in an enum `enum F {X = 'X'}`)

2. If you have an object type `type O = {a: {b: {c: string}}}`, what's the type of `keyof O`? What about `O['a']['b']`?
3. Write an `Exclusive<T, U>` type that computes the types that are in either `T` or `U`, but not both. For example, `Exclusive<1 | 2 | 3, 2 | 3 | 4>` should resolve to `1 | 4`. Write out step by step how the typechecker evaluates `Exclusive<1 | 2, 2 | 4>`.
4. Rewrite the example (from "Definite Assignment Assertions") to avoid the definite assignment assertion.

1 Symbolic execution is a form of program analysis where you use a special program called a symbolic evaluator to run your program the same way a runtime would, but without assigning definite values to variables; instead, each variable is modelled as a *symbol* whose value gets constrained as the program runs. Symbolic execution lets you say things like "this variable is never used," or "this function never returns," or "in the positive branch of the `if` statement on line 102, variable `x` is guaranteed not to be `null`."

2 Flow-based type inference is supported by a handful of languages, including TypeScript, Flow, Kotlin, and Ceylon. It's a way to refine types within a block of code, and is an alternative to C/Java-style explicit type annotations and Haskell/OCaml/Scala-style pattern matching. The idea is to take a symbolic execution engine and embed it right in the typechecker, in order to give feedback to the typechecker and reason through a program in a way that is closer to how a human programmer might do it.

3 JavaScript has seven falsy values: `null`, `undefined`, `NaN`, `0`, `-0`, `""`, and of course, `false`. Everything else is truthy.

4 DefinitelyTyped is the open source repository for type declarations for third-party JavaScript. To learn more, jump ahead to "JavaScript That Has Type Declarations on DefinitelyTyped".

5 In some languages, these are also called *opaque types*.

6 There are other reasons why you might want to avoid extending the prototype, like code portability, making your dependency graphs more explicit, or improving performance by only loading those methods that you actually use. However, safety is no longer one of those reasons.