
TypeScript's Type System

TypeScript generates code ([Item 3](#)), but the type system is the main event. This is why you're using the language!

This chapter walks you through the nuts and bolts of TypeScript's type system: how to think about it, how to use it, choices you'll need to make, and features you should avoid. TypeScript's type system is surprisingly powerful and able to express things you might not expect a type system to be able to. The items in this chapter will give you a solid foundation to build upon as you write TypeScript and read the rest of this book.

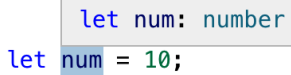
Item 6: Use Your Editor to Interrogate and Explore the Type System

When you install TypeScript, you get two executables:

- `tsc`, the TypeScript compiler
- `tsserver`, the TypeScript standalone server

You're much more likely to run the TypeScript compiler directly, but the server is every bit as important because it provides *language services*. These include autocomplete, inspection, navigation, and refactoring. You typically use these services through your editor. If yours isn't configured to provide them, then you're missing out! Services like autocomplete are one of the things that make TypeScript such a joy to use. But beyond convenience, your editor is the best place to build and test your knowledge of the type system. This will help you build an intuition for when TypeScript is able to infer types, which is key to writing compact, idiomatic code (see [Item 19](#)).

The details will vary from editor to editor, but you can generally mouse over a symbol to see what TypeScript considers its type (see [Figure 2-1](#)).



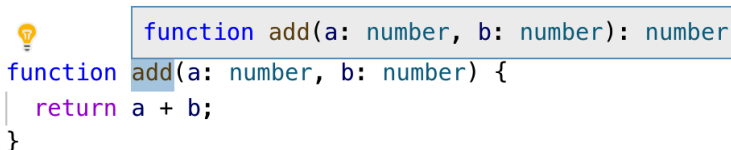
```
let num: number
let num = 10;
```

A screenshot of a code editor showing two lines of TypeScript code. The first line is `let num: number` and the second line is `let num = 10;`. A tooltip is displayed over the `num` variable in the second line, showing its inferred type as `number`.

Figure 2-1. An editor (vscode) showing that the inferred type of the `num` symbol is `number`

You didn't write `number` here, but TypeScript was able to figure it out based on the value `10`.

You can also inspect functions, as shown in [Figure 2-2](#).



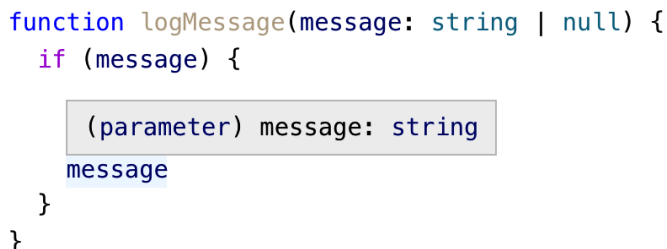
```
function add(a: number, b: number): number {
  return a + b;
}
```

A screenshot of a code editor showing a function definition. The function is `function add(a: number, b: number): number { return a + b; }`. A tooltip is displayed over the function signature, showing the inferred return type as `number`.

Figure 2-2. Using an editor to reveal the inferred type for a function

The noteworthy bit of information is the inferred value for the return type, `number`. If this does not match your expectation, you should add a type declaration and track down the discrepancy (see [Item 9](#)).

Seeing TypeScript's understanding of a variable's type at any given point is essential for building an intuition around widening ([Item 21](#)) and narrowing ([Item 22](#)). Seeing the type of a variable change in the branch of a conditional is a tremendous way to build confidence in the type system (see [Figure 2-3](#)).



```
function logMessage(message: string | null) {
  if (message) {
    (parameter) message: string
    message
  }
}
```

A screenshot of a code editor showing a function `logMessage` with a parameter `message` of type `string | null`. Inside the function, there is an `if (message)` branch. A tooltip is displayed over the `message` variable inside the branch, showing its inferred type as `string`.

Figure 2-3. The type of `message` is `string | null` outside the branch but `string` inside.

You can inspect individual properties in a larger object to see what TypeScript has inferred about them (see [Figure 2-4](#)).

```
const foo = {  
  (property) x: number[]  
  x: [1, 2, 3],  
  bar: {  
    name: 'Fred'  
  }  
};
```

Figure 2-4. Inspecting how TypeScript has inferred types in an object

If your intention was for `x` to be a tuple type (`[number, number, number]`), then a type annotation will be required.

To see inferred generic types in the middle of a chain of operations, inspect the method name (as shown in [Figure 2-5](#)).

```
function restOfPath(path: string) {  
  (method) Array<string>.slice(start?: number, end?: number): string[]  
  Returns a section of an array.  
  @param start — The beginning of the specified portion of the array.  
  @param end — The end of the specified portion of the array.  
  return path.split('/').slice(1).join('/');  
}
```

Figure 2-5. Revealing inferred generic types in a chain of method calls

The `Array<string>` indicates that TypeScript understands that `split` produced an array of strings. While there was little ambiguity in this case, this information can prove essential in writing and debugging long chains of function calls.

Seeing type errors in your editor can also be a great way to learn the nuances of the type system. For example, this function tries to get an `HTMLElement` by its ID, or return a default one. TypeScript flags two errors:

```
function getElement(eId: string | HTMLElement | null): HTMLElement {  
  if (typeof eId === 'object') {  
    return eId;  
  }  
  // ~~~~~ 'HTMLElement | null' is not assignable to 'HTMLElement'  
  } else if (eId === null) {  
    return document.body;  
  } else {  
    const el = document.getElementById(eId);  
  }  
}
```

```

    return el;
    // ~~~~~ 'HTMLElement | null' is not assignable to 'HTMLElement'
  }
}

```

The intent in the first branch of the `if` statement was to filter down to just the objects, namely, the `HTMLElement`s. But oddly enough, in JavaScript `typeof null` is `"object"`, so `elOrId` could still be `null` in that branch. You can fix this by putting the `null` check first. The second error is because `document.getElementById` can return `null`, so you need to handle that case as well, perhaps by throwing an exception.

Language services can also help you navigate through libraries and type declarations. Suppose you see a call to the `fetch` function in code and want to learn more about it. Your editor should provide a “Go to Definition” option. In mine it looks like it does in [Figure 2-6](#).

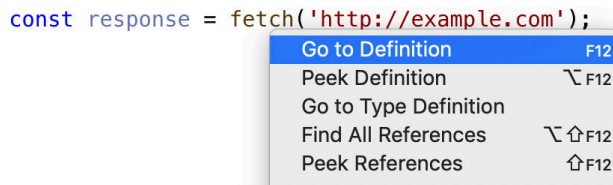


Figure 2-6. The TypeScript language service provides a “Go to Definition” feature that should be surfaced in your editor.

Selecting this option takes you into `lib.dom.d.ts`, the type declarations which TypeScript includes for the DOM:

```

declare function fetch(
  input: RequestInfo, init?: RequestInit
): Promise<Response>;

```

You can see that `fetch` returns a `Promise` and takes two arguments. Clicking through on `RequestInfo` brings you here:

```

type RequestInfo = Request | string;

```

from which you can go to `Request`:

```

declare var Request: {
  prototype: Request;
  new(input: RequestInfo, init?: RequestInit): Request;
};

```

Here you can see that the `Request` type and value are being modeled separately (see [Item 8](#)). You’ve seen `RequestInfo` already. Clicking through on `RequestInit` shows everything you can use to construct a `Request`:

```
interface RequestInit {
  body?: BodyInit | null;
  cache?: RequestCache;
  credentials?: RequestCredentials;
  headers?: HeadersInit;
  // ...
}
```

There are many more types you could follow here, but you get the idea. Type declarations can be challenging to read at first, but they're an excellent way to see what can be done with TypeScript, how the library you're using is modeled, and how you might debug errors. For much more on type declarations, see [Chapter 6](#).

Things to Remember

- Take advantage of the TypeScript language services by using an editor that can use them.
- Use your editor to build an intuition for how the type system works and how TypeScript infers types.
- Know how to jump into type declaration files to see how they model behavior.

Item 7: Think of Types as Sets of Values

At runtime, every variable has a single value chosen from JavaScript's universe of values. There are many possible values, including:

- 42
- null
- undefined
- 'Canada'
- {animal: 'Whale', weight_lbs: 40_000}
- /regex/
- new HTMLButtonElement
- (x, y) => x + y

But before your code runs, when TypeScript is checking it for errors, it just has a *type*. This is best thought of as a *set of possible values*. This set is known as the *domain* of the type. For instance, you can think of the *number* type as the set of all number values. 42 and -37.25 are in it, but 'Canada' is not. Depending on `strictNullChecks`, `null` and `undefined` may or may not be part of the set.

The smallest set is the empty set, which contains no values. It corresponds to the `never` type in TypeScript. Because its domain is empty, no values are assignable to a variable with a `never` type:

```
const x: never = 12;
// ~ Type '12' is not assignable to type 'never'
```

The next smallest sets are those which contain single values. These correspond to literal types in TypeScript, also known as unit types:

```
type A = 'A';
type B = 'B';
type Twelve = 12;
```

To form types with two or three values, you can union unit types:

```
type AB = 'A' | 'B';
type AB12 = 'A' | 'B' | 12;
```

and so on. Union types correspond to unions of sets of values.

The word “assignable” appears in many TypeScript errors. In the context of sets of values, it means either “member of” (for a relationship between a value and a type) or “subset of” (for a relationship between two types):

```
const a: AB = 'A'; // OK, value 'A' is a member of the set {'A', 'B'}
const c: AB = 'C';
// ~ Type '"C"' is not assignable to type 'AB'
```

The type `"C"` is a unit type. Its domain consists of the single value `"C"`. This is not a subset of the domain of `AB` (which consists of the values `"A"` and `"B"`), so this is an error. At the end of the day, almost all the type checker is doing is testing whether one set is a subset of another:

```
// OK, {"A", "B"} is a subset of {"A", "B"}:
const ab: AB = Math.random() < 0.5 ? 'A' : 'B';
const ab12: AB12 = ab; // OK, {"A", "B"} is a subset of {"A", "B", 12}

declare let twelve: AB12;
const back: AB = twelve;
// ~~~~ Type 'AB12' is not assignable to type 'AB'
//      Type '12' is not assignable to type 'AB'
```

The sets for these types are easy to reason about because they are finite. But most types that you work with in practice have infinite domains. Reasoning about these can be harder. You can think of them as either being built constructively:

```
type Int = 1 | 2 | 3 | 4 | 5 // / ...
```

or by describing their members:

```
interface Identified {
  id: string;
}
```

Think of this interface as a description of the values in the domain of its type. Does the value have an `id` property whose value is assignable to (a member of) `string`? Then it's an `Identifiable`.

That's *all* it says. As [Item 4](#) explained, TypeScript's structural typing rules mean that the value could have other properties, too. It could even be callable! This fact can sometimes be obscured by excess property checking (see [Item 11](#)).

Thinking of types as sets of values helps you reason about operations on them. For example:

```
interface Person {
  name: string;
}
interface Lifespan {
  birth: Date;
  death?: Date;
}
type PersonSpan = Person & Lifespan;
```

The `&` operator computes the intersection of two types. What sorts of values belong to the `PersonSpan` type? On first glance the `Person` and `Lifespan` interfaces have no properties in common, so you might expect it to be the empty set (i.e., the never type). But type operations apply to the sets of values (the domain of the type), not to the properties in the interface. And remember that values with additional properties still belong to a type. So a value that has the properties of *both* `Person` *and* `Lifespan` will belong to the intersection type:

```
const ps: PersonSpan = {
  name: 'Alan Turing',
  birth: new Date('1912/06/23'),
  death: new Date('1954/06/07'),
}; // OK
```

Of course, a value could have more than those three properties and still belong to the type! The general rule is that values in an intersection type contain the union of properties in each of its constituents.

The intuition about intersecting properties is correct, but for the *union* of two interfaces, rather than their intersection:

```
type K = keyof (Person | Lifespan); // Type is never
```

There are no keys that TypeScript can guarantee belong to a value in the union type, so `keyof` for the union must be the empty set (never). Or, more formally:

```
keyof (A&B) = (keyof A) | (keyof B)
keyof (A|B) = (keyof A) & (keyof B)
```

If you can build an intuition for why these equations hold, you'll have come a long way toward understanding TypeScript's type system!

Another perhaps more common way to write the `PersonSpan` type would be with `extends`:

```
interface Person {  
  name: string;  
}  
interface PersonSpan extends Person {  
  birth: Date;  
  death?: Date;  
}
```

Thinking of types as sets of values, what does `extends` mean? Just like “assignable to,” you can read it as “subset of.” Every value in `PersonSpan` must have a `name` property which is a `string`. And every value must also have a `birth` property, so it’s a proper subset.

You might hear the term “subtype.” This is another way of saying that one set’s domain is a subset of the others. Thinking in terms of one-, two-, and three-dimensional vectors:

```
interface Vector1D { x: number; }  
interface Vector2D extends Vector1D { y: number; }  
interface Vector3D extends Vector2D { z: number; }
```

You’d say that a `Vector3D` is a subtype of `Vector2D`, which is a subtype of `Vector1D` (in the context of classes you’d say “subclass”). This relationship is usually drawn as a hierarchy, but thinking in terms of sets of values, a Venn diagram is more appropriate (see [Figure 2-7](#)).

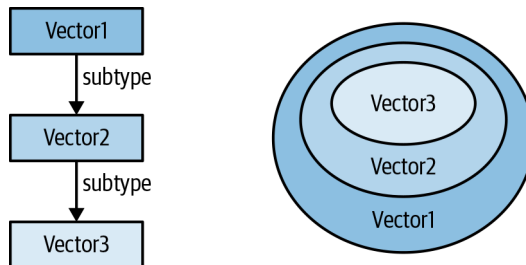


Figure 2-7. Two ways of thinking of type relationships: as a hierarchy or as overlapping sets

With the Venn diagram, it’s clear that the subset/subtype/assignability relationships are unchanged if you rewrite the interfaces without `extends`:

```
interface Vector1D { x: number; }  
interface Vector2D { x: number; y: number; }  
interface Vector3D { x: number; y: number; z: number; }
```

The sets haven’t changed, so neither has the Venn diagram.

While both interpretations are workable for object types, the set interpretation becomes much more intuitive when you start thinking about literal types and union types. `extends` can also appear as a constraint in a generic type, and it also means “subset of” in this context (Item 14):

```
function getKey<K extends string>(val: any, key: K) {  
    // ...  
}
```

What does it mean to extend `string`? If you’re used to thinking in terms of object inheritance, it’s hard to interpret. You could define a subclass of the object wrapper type `String` (Item 10), but that seems inadvisable.

Thinking in terms of sets, on the other hand, it’s crystal clear: any type whose domain is a subset of `string` will do. This includes string literal types, unions of string literal types and `string` itself:

```
getKey({}, 'x'); // OK, 'x' extends string  
getKey({}, Math.random() < 0.5 ? 'a' : 'b'); // OK, 'a'|'b' extends string  
getKey({}, document.title); // OK, string extends string  
getKey({}, 12);  
// ~~ Type '12' is not assignable to parameter of type 'string'
```

“extends” has turned into “assignable” in the last error, but this shouldn’t trip us up since we know to read both as “subset of.” This is also a helpful mindset with finite sets, such the ones you might get from `keyof T`, which returns type for just the keys of an object type:

```
interface Point {  
    x: number;  
    y: number;  
}  
type PointKeys = keyof Point; // Type is "x" | "y"  
  
function sortBy<K extends keyof T, T>(vals: T[], key: K): T[] {  
    // ...  
}  
  
const pts: Point[] = [{x: 1, y: 1}, {x: 2, y: 0}];  
sortBy(pts, 'x'); // OK, 'x' extends 'x'|'y' (aka keyof T)  
sortBy(pts, 'y'); // OK, 'y' extends 'x'|'y'  
sortBy(pts, Math.random() < 0.5 ? 'x' : 'y'); // OK, 'x'|'y' extends 'x'|'y'  
sortBy(pts, 'z');  
// ~~~ Type '"z"' is not assignable to parameter of type '"x" | "y"'
```

The set interpretation also makes more sense when you have types whose relationship isn’t strictly hierarchical. What’s the relationship between `string|number` and `string|Date`, for instance? Their intersection is non-empty (it’s `string`), but neither is a subset of the other. The relationship between their domains is clear, even though these types don’t fit into a strict hierarchy (see Figure 2-8).

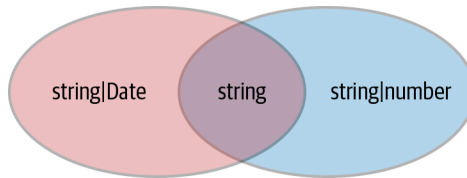


Figure 2-8. Union types may not fit into a hierarchy but can be thought of in terms of sets of values.

Thinking of types as sets can also clarify the relationships between arrays and tuples. For example:

```
const list = [1, 2]; // Type is number[]
const tuple: [number, number] = list;
// ~~~~ Type 'number[]' is missing the following
//      properties from type '[number, number]': 0, 1
```

Are there lists of numbers which are not pairs of numbers? Sure! The empty list and the list `[1]` are examples. It therefore makes sense that `number[]` is not assignable to `[number, number]` since it's not a subset of it. (The reverse assignment does work.)

Is a triple assignable to a pair? Thinking in terms of structural typing, you might expect it to be. A pair has 0 and 1 keys, so mightn't it have others, too, like 2?

```
const triple: [number, number, number] = [1, 2, 3];
const double: [number, number] = triple;
// ~~~~~ '[number, number, number]' is not assignable to '[number, number]'
//      Types of property 'length' are incompatible
//      Type '3' is not assignable to type '2'
```

The answer is “no,” and for an interesting reason. Rather than modeling a pair of numbers as `{0: number, 1: number}`, TypeScript models it as `{0: number, 1: number, length: 2}`. This makes sense—you can check the length of a tuple—and it precludes this assignment. And that's probably for the best!

If types are best thought of as sets of values, that means that two types with the same sets of values are the same. And indeed this is true. Unless two types are semantically different and just happen to have the same domain, there's no reason to define the same type twice.

Finally, it's worth noting that not all sets of values correspond to TypeScript types. There is no TypeScript type for all the integers, or for all the objects that have `x` and `y` properties but no others. You can sometimes subtract types using `Exclude`, but only when it would result in a proper TypeScript type:

```
type T = Exclude<string|Date, string|number>; // Type is Date
type NonZeroNums = Exclude<number, 0>; // Type is still just number
```

Table 2-1 summarizes the correspondence between TypeScript terms and terms from set theory.

Table 2-1. TypeScript terms and set terms

| TypeScript term | Set term |
|-----------------------|----------------------------------|
| never | \emptyset (empty set) |
| Literal type | Single element set |
| Value assignable to T | $\text{Value} \in T$ (member of) |
| T1 assignable to T2 | $T1 \subseteq T2$ (subset of) |
| T1 extends T2 | $T1 \subseteq T2$ (subset of) |
| $T1 \mid T2$ | $T1 \cup T2$ (union) |
| $T1 \& T2$ | $T1 \cap T2$ (intersection) |
| unknown | Universal set |

Things to Remember

- Think of types as sets of values (the type’s *domain*). These sets can either be finite (e.g., boolean or literal types) or infinite (e.g., number or string).
- TypeScript types form intersecting sets (a Venn diagram) rather than a strict hierarchy. Two types can overlap without either being a subtype of the other.
- Remember that an object can still belong to a type even if it has additional properties that were not mentioned in the type declaration.
- Type operations apply to a set’s domain. The intersection of A and B is the intersection of A’s domain and B’s domain. For object types, this means that values in $A \& B$ have the properties of both A and B.
- Think of “extends,” “assignable to,” and “subtype of” as synonyms for “subset of.”

Item 8: Know How to Tell Whether a Symbol Is in the Type Space or Value Space

A symbol in TypeScript exists in one of two spaces:

- Type space
- Value space

This can get confusing because the same name can refer to different things depending on which space it’s in:

```
interface Cylinder {
  radius: number;
  height: number;
}
```

```
const Cylinder = (radius: number, height: number) => ({radius, height});
```

interface `Cylinder` introduces a symbol in type space. `const Cylinder` introduces a symbol with the same name in value space. They have nothing to do with one another. Depending on the context, when you type `Cylinder`, you'll either be referring to the type or the value. Sometimes this can lead to errors:

```
function calculateVolume(shape: unknown) {
  if (shape instanceof Cylinder) {
    shape.radius
    // ~~~~~ Property 'radius' does not exist on type '{}'
  }
}
```

What's going on here? You probably intended the `instanceof` to check whether the shape was of the `Cylinder` type. But `instanceof` is JavaScript's runtime operator, and it operates on values. So `instanceof Cylinder` refers to the function, not the type.

It's not always obvious at first glance whether a symbol is in type space or value space. You have to tell from the context in which the symbol occurs. This can get especially confusing because many type-space constructs look exactly the same as value-space constructs.

Literals, for example:

```
type T1 = 'string literal';
type T2 = 123;
const v1 = 'string literal';
const v2 = 123;
```

Generally the symbols after a type or interface are in type space while those introduced in a `const` or `let` declaration are values.

One of the best ways to build an intuition for the two spaces is through the [TypeScript Playground](#), which shows you the generated JavaScript for your TypeScript source. Types are erased during compilation ([Item 3](#)), so if a symbol disappears then it was probably in type space (see [Figure 2-9](#)).

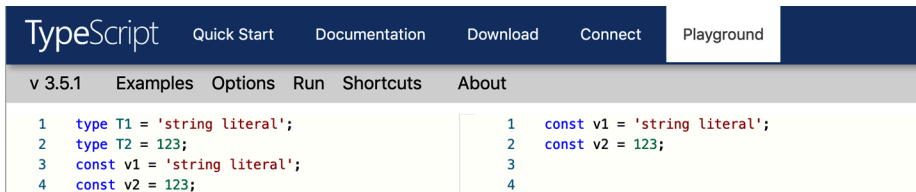


Figure 2-9. The TypeScript playground showing generated JavaScript. The symbols on the first two lines go away, so they were in type space.

Statements in TypeScript can alternate between type space and value space. The symbols after a type declaration (`:`) or an assertion (`as`) are in type space while everything after an `=` is in value space. For example:

```
interface Person {
  first: string;
  last: string;
}
const p: Person = { first: 'Jane', last: 'Jacobs' };
//      - ----- Values
//      ----- Type
```

Function statements in particular can alternate repeatedly between the spaces:

```
function email(p: Person, subject: string, body: string): Response {
  //      ----- Values
  //      ----- Types
  // ...
}
```

The class and enum constructs introduce both a type and a value. In the first example, `Cylinder` should have been a class:

```
class Cylinder {
  radius=1;
  height=1;
}

function calculateVolume(shape: unknown) {
  if (shape instanceof Cylinder) {
    shape // OK, type is Cylinder
    shape.radius // OK, type is number
  }
}
```

The TypeScript type introduced by a class is based on its shape (its properties and methods) while the value is the constructor.

There are many operators and keywords that mean different things in a type or value context. `typeof`, for instance:

```

type T1 = typeof p; // Type is Person
type T2 = typeof email;
// Type is (p: Person, subject: string, body: string) => Response

const v1 = typeof p; // Value is "object"
const v2 = typeof email; // Value is "function"

```

In a type context, `typeof` takes a value and returns its TypeScript type. You can use these as part of a larger type expression, or use a `type` statement to give them a name.

In a value context, `typeof` is JavaScript’s runtime `typeof` operator. It returns a string containing the runtime type of the symbol. This is *not* the same as the TypeScript type! JavaScript’s runtime type system is much simpler than TypeScript’s static type system. In contrast to the infinite variety of TypeScript types, there have historically only been six runtime types in JavaScript: “string,” “number,” “boolean,” “undefined,” “object,” and “function.”

`typeof` always operates on values. You can’t apply it to types. The `class` keyword introduces both a value and a type, so what is the `typeof` a class? It depends on the context:

```

const v = typeof Cylinder; // Value is "function"
type T = typeof Cylinder; // Type is typeof Cylinder

```

The value is “function” because of how classes are implemented in JavaScript. The type isn’t particularly illuminating. What’s important is that it’s *not* `Cylinder` (the type of an instance). It’s actually the constructor function, which you can see by using it with `new`:

```

declare let fn: T;
const c = new fn(); // Type is Cylinder

```

You can go between the constructor type and the instance type using the `InstanceType` generic:

```

type C = InstanceType<typeof Cylinder>; // Type is Cylinder

```

The `[]` property accessor also has an identical-looking equivalent in type space. But be aware that while `obj['field']` and `obj.field` are equivalent in value space, they are not in type space. You must use the former to get the type of another type’s property:

```

const first: Person['first'] = p['first']; // Or p.first
// ----- Values
// ----- Types

```

`Person['first']` is a *type* here since it appears in a type context (after a `:`). You can put any type in the index slot, including union types or primitive types:

```

type PersonEl = Person['first' | 'last']; // Type is string
type Tuple = [string, number, Date];
type TupleEl = Tuple[number]; // Type is string | number | Date

```

See [Item 14](#) for more on this.

There are many other constructs that have different meanings in the two spaces:

- `this` in value space is JavaScript’s `this` keyword ([Item 49](#)). As a type, `this` is the TypeScript type of `this`, aka “polymorphic `this`.” It’s helpful for implementing method chains with subclasses.
- In value space `&` and `|` are bitwise AND and OR. In type space they are the intersection and union operators.
- `const` introduces a new variable, but as `const` changes the inferred type of a literal or literal expression ([Item 21](#)).
- `extends` can define a subclass (`class A extends B`) or a subtype (`interface A extends B`) or a constraint on a generic type (`Generic<T extends number>`).
- `in` can either be part of a loop (`for (key in object)`) or a mapped type ([Item 14](#)).

If TypeScript doesn’t seem to understand your code at all, it may be because of confusion around type and value space. For example, say you change the `email` function from earlier to take its arguments in a single object parameter:

```

function email(options: {person: Person, subject: string, body: string}) {
  // ...
}

```

In JavaScript you can use destructuring assignment to create local variables for each property in the object:

```

function email({person, subject, body}) {
  // ...
}

```

If you try to do the same in TypeScript, you get some confusing errors:

```

function email({
  person: Person,
  // ~~~~~ Binding element 'Person' implicitly has an 'any' type
  subject: string,
  // ~~~~~ Duplicate identifier 'string'
  // ~~~~~ Binding element 'string' implicitly has an 'any' type
  body: string}
  // ~~~~~ Duplicate identifier 'string'
  // ~~~~~ Binding element 'string' implicitly has an 'any' type
) { /* ... */ }

```

The problem is that `Person` and `string` are being interpreted in a value context. You're trying to create a variable named `Person` and two variables named `string`. Instead, you should separate the types and values:

```
function email(  
  {person, subject, body}: {person: Person, subject: string, body: string}  
) {  
  // ...  
}
```

This is significantly more verbose, but in practice you may have a named type for the parameters or be able to infer them from context (Item 26).

While the similar constructs in type and value can be confusing at first, they're eventually useful as a mnemonic once you get the hang of it.

Things to Remember

- Know how to tell whether you're in type space or value space while reading a TypeScript expression. Use the TypeScript playground to build an intuition for this.
- Every value has a type, but types do not have values. Constructs such as `type` and `interface` exist only in the type space.
- `"foo"` might be a string literal, or it might be a string literal type. Be aware of this distinction and understand how to tell which it is.
- `typeof`, `this`, and many other operators and keywords have different meanings in type space and value space.
- Some constructs such as `class` or `enum` introduce both a type and a value.

Item 9: Prefer Type Declarations to Type Assertions

TypeScript seems to have two ways of assigning a value to a variable and giving it a type:

```
interface Person { name: string };  
  
const alice: Person = { name: 'Alice' }; // Type is Person  
const bob = { name: 'Bob' } as Person; // Type is Person
```

While these achieve similar ends, they are actually quite different! The first (`alice: Person`) adds a *type declaration* to the variable and ensures that the value conforms to the type. The latter (`as Person`) performs a *type assertion*. This tells TypeScript that, despite the type it inferred, you know better and would like the type to be `Person`.

In general, you should prefer type declarations to type assertions. Here's why:


```
const alice: Person = {};
// ~~~~~ Property 'name' is missing in type '{}'
//       but required in type 'Person'
const bob = {} as Person; // No error
```

The type declaration verifies that the value conforms to the interface. Since it does not, TypeScript flags an error. The type assertion silences this error by telling the type checker that, for whatever reason, you know better than it does.

The same thing happens if you specify an additional property:

```
const alice: Person = {
  name: 'Alice',
  occupation: 'TypeScript developer'
// ~~~~~~ Object literal may only specify known properties
//       and 'occupation' does not exist in type 'Person'
};
const bob = {
  name: 'Bob',
  occupation: 'JavaScript developer'
} as Person; // No error
```

This is excess property checking at work ([Item 11](#)), but it doesn't apply if you use an assertion.

Because they provide additional safety checks, you should use type declarations unless you have a specific reason to use a type assertion.



You may also see code that looks like `const bob = <Person>{}`. This was the original syntax for assertions and is equivalent to `{}` `as Person`. It is less common now because `<Person>` is interpreted as a start tag in `.tsx` files (TypeScript + React).

It's not always clear how to use a declaration with arrow functions. For example, what if you wanted to use the named `Person` interface in this code?

```
const people = ['alice', 'bob', 'jan'].map(name => ({name}));
// { name: string; }[]... but we want Person[]
```

It's tempting to use a type assertion here, and it seems to solve the problem:

```
const people = ['alice', 'bob', 'jan'].map(
  name => ({name} as Person)
); // Type is Person[]
```

But this suffers from all the same issues as a more direct use of type assertions. For example:

```
const people = ['alice', 'bob', 'jan'].map(name => ({} as Person));
// No error
```

So how do you use a type declaration in this context instead? The most straightforward way is to declare a variable in the arrow function:

```
const people = ['alice', 'bob', 'jan'].map(name => {
  const person: Person = {name};
  return person
}); // Type is Person[]
```

But this introduces considerable noise compared to the original code. A more concise way is to declare the return type of the arrow function:

```
const people = ['alice', 'bob', 'jan'].map(
  (name): Person => ({name})
); // Type is Person[]
```

This performs all the same checks on the value as the previous version. The parentheses are significant here! `(name): Person` infers the type of `name` and specifies that the returned type should be `Person`. But `(name: Person)` would specify the type of `name` as `Person` and allow the return type to be inferred, which would produce an error.

In this case you could have also written the final desired type and let TypeScript check the validity of the assignment:

```
const people: Person[] = ['alice', 'bob', 'jan'].map(
  (name): Person => ({name})
);
```

But in the context of a longer chain of function calls it may be necessary or desirable to have the named type in place earlier. And it will help flag errors where they occur.

So when *should* you use a type assertion? Type assertions make the most sense when you truly do know more about a type than TypeScript does, typically from context that isn't available to the type checker. For instance, you may know the type of a DOM element more precisely than TypeScript does:

```
document.querySelector('#myButton').addEventListener('click', e => {
  e.currentTarget // Type is EventTarget
  const button = e.currentTarget as HTMLButtonElement;
  button // Type is HTMLButtonElement
});
```

Because TypeScript doesn't have access to the DOM of your page, it has no way of knowing that `#myButton` is a button element. And it doesn't know that the current Target of the event should be that same button. Since you have information that TypeScript does not, a type assertion makes sense here. For more on DOM types, see [Item 55](#).

You may also run into the non-null assertion, which is so common that it gets a special syntax:

```
const elNull = document.getElementById('foo'); // Type is HTMLElement | null
const el = document.getElementById('foo')!; // Type is HTMLElement
```

Used as a prefix, `!` is boolean negation. But as a suffix, `!` is interpreted as an assertion that the value is non-null. You should treat `!` just like any other assertion: it is erased during compilation, so you should only use it if you have information that the type checker lacks and can ensure that the value is non-null. If you can't, you should use a conditional to check for the `null` case.

Type assertions have their limits: they don't let you convert between arbitrary types. The general idea is that you can use a type assertion to convert between `A` and `B` if either is a subset of the other. `HTMLElement` is a subtype of `HTMLElement | null`, so this type assertion is OK. `HTMLButtonElement` is a subtype of `EventTarget`, so that was OK, too. And `Person` is a subtype of `{}`, so that assertion is also fine.

But you can't convert between a `Person` and an `HTMLElement` since neither is a subtype of the other:

```
interface Person { name: string; }
const body = document.body;
const el = body as Person;
// ~~~~~ Conversion of type 'HTMLElement' to type 'Person'
//         may be a mistake because neither type sufficiently
//         overlaps with the other. If this was intentional,
//         convert the expression to 'unknown' first
```

The error suggests an escape hatch, namely, using the unknown type ([Item 42](#)). Every type is a subtype of `unknown`, so assertions involving `unknown` are always OK. This lets you convert between arbitrary types, but at least you're being explicit that you're doing something suspicious!

```
const el = document.body as unknown as Person; // OK
```

Things to Remember

- Prefer type declarations (`: Type`) to type assertions (`as Type`).
- Know how to annotate the return type of an arrow function.
- Use type assertions and non-null assertions when you know something about types that TypeScript does not.

Item 10: Avoid Object Wrapper Types (String, Number, Boolean, Symbol, BigInt)

In addition to objects, JavaScript has seven types of primitive values: strings, numbers, booleans, `null`, `undefined`, symbol, and `bigint`. The first five have been around since the beginning. The symbol primitive was added in ES2015, and `bigint` is in the process of being finalized.

Primitives are distinguished from objects by being immutable and not having methods. You might object that strings *do* have methods:

```
> 'primitive'.charAt(3)
"n"
```

But things are not quite as they seem. There's actually something surprising and subtle going on here. While a string *primitive* does not have methods, JavaScript also defines a `String` *object* type that does. JavaScript freely converts between these types. When you access a method like `charAt` on a string primitive, JavaScript wraps it in a `String` object, calls the method, and then throws the object away.

You can observe this if you monkey-patch `String.prototype` (Item 43):

```
// Don't do this!
const originalCharAt = String.prototype.charAt;
String.prototype.charAt = function(pos) {
  console.log(this, typeof this, pos);
  return originalCharAt.call(this, pos);
};
console.log('primitive'.charAt(3));
```

This produces the following output:

```
[String: 'primitive'] 'object' 3
n
```

The `this` value in the method is a `String` object wrapper, not a string primitive. You can instantiate a `String` object directly and it will sometimes behave like a string primitive. But not always. For example, a `String` object is only ever equal to itself:

```
> "hello" === new String("hello")
false
> new String("hello") === new String("hello")
false
```

The implicit conversion to object wrapper types explains an odd phenomenon in JavaScript—if you assign a property to a primitive, it disappears:

```
> x = "hello"
> x.language = 'English'
'English'
> x.language
undefined
```

Now you know the explanation: `x` is converted to a `String` instance, the `language` property is set on that, and then the object (with its `language` property) is thrown away.

There are object wrapper types for the other primitives as well: `Number` for numbers, `Boolean` for booleans, `Symbol` for symbols, and `BigInt` for bigints (there are no object wrappers for `null` and `undefined`).

These wrapper types exist as a convenience to provide methods on the primitive values and to provide static methods (e.g., `String.fromCharCode`). But there's usually no reason to instantiate them directly.

TypeScript models this distinction by having distinct types for the primitives and their object wrappers:

- `string` and `String`
- `number` and `Number`
- `boolean` and `Boolean`
- `symbol` and `Symbol`
- `bigint` and `BigInt`

It's easy to inadvertently type `String` (especially if you're coming from Java or C#) and it even seems to work, at least initially:

```
function getStringLen(foo: String) {  
    return foo.length;  
}  
  
getStringLen("hello"); // OK  
getStringLen(new String("hello")); // OK
```

But things go awry when you try to pass a `String` object to a method that expects a `string`:

```
function isGreeting(phrase: String) {  
    return [  
        'hello',  
        'good day'  
    ].includes(phrase);  
    // ~~~~~  
    // Argument of type 'String' is not assignable to parameter  
    // of type 'string'.  
    // 'string' is a primitive, but 'String' is a wrapper object;  
    // prefer using 'string' when possible  
}
```

So `string` is assignable to `String`, but `String` is not assignable to `string`. Confusing? Follow the advice in the error message and stick with `string`. All the type declarations that ship with TypeScript use it, as do the typings for almost all other libraries.

Another way you can wind up with wrapper objects is if you provide an explicit type annotation with a capital letter:

```
const s: String = "primitive";  
const n: Number = 12;  
const b: Boolean = true;
```

Of course, the values at runtime are still primitives, not objects. But TypeScript permits these declarations because the primitive types are assignable to the object wrappers. These annotations are both misleading and redundant (Item 19). Better to stick with the primitive types.

As a final note, it's OK to call `BigInt` and `Symbol` without `new`, since these create primitives:

```
> typeof BigInt(1234)
"bigint"
> typeof Symbol('sym')
"symbol"
```

These are the `BigInt` and `Symbol` *values*, not the TypeScript types (Item 8). Calling them results in values of type `bigint` and `symbol`.

Things to Remember

- Understand how object wrapper types are used to provide methods on primitive values. Avoid instantiating them or using them directly.
- Avoid TypeScript object wrapper types. Use the primitive types instead: `string` instead of `String`, `number` instead of `Number`, `boolean` instead of `Boolean`, `symbol` instead of `Symbol`, and `bigint` instead of `BigInt`.

Item 11: Recognize the Limits of Excess Property Checking

When you assign an object literal to a variable with a declared type, TypeScript makes sure it has the properties of that type *and no others*:

```
interface Room {
  numDoors: number;
  ceilingHeightFt: number;
}
const r: Room = {
  numDoors: 1,
  ceilingHeightFt: 10,
  elephant: 'present',
// ~~~~~ Object literal may only specify known properties,
//         and 'elephant' does not exist in type 'Room'
};
```

While it is odd that there's an `elephant` property, this error doesn't make much sense from a structural typing point of view (Item 4). That constant *is* assignable to the `Room` type, which you can see by introducing an intermediate variable:

```
const obj = {
  numDoors: 1,
```

```

    ceilingHeightFt: 10,
    elephant: 'present',
  };
  const r: Room = obj; // OK

```

The type of `obj` is inferred as `{ numDoors: number; ceilingHeightFt: number; elephant: string }`. Because this type includes a subset of the values in the `Room` type, it is assignable to `Room`, and the code passes the type checker (see [Item 7](#)).

So what is different about these two examples? In the first you’ve triggered a process known as “excess property checking,” which helps catch an important class of errors that the structural type system would otherwise miss. But this process has its limits, and conflating it with regular assignability checks can make it harder to build an intuition for structural typing. Recognizing excess property checking as a distinct process will help you build a clearer mental model of TypeScript’s type system.

As [Item 1](#) explained, TypeScript goes beyond trying to flag code that will throw exceptions at runtime. It also tries to find code that doesn’t do what you intend. Here’s an example of the latter:

```

interface Options {
  title: string;
  darkMode?: boolean;
}
function createWindow(options: Options) {
  if (options.darkMode) {
    setDarkMode();
  }
  // ...
}
createWindow({
  title: 'Spider Solitaire',
  darkmode: true
// ~~~~~ Object literal may only specify known properties, but
//         'darkmode' does not exist in type 'Options'.
//         Did you mean to write 'darkMode'?
});

```

This code doesn’t throw any sort of error at runtime. But it’s also unlikely to do what you intended for the exact reason that TypeScript says: it should be `darkMode` (capital M), not `darkmode`.

A purely structural type checker wouldn’t be able to spot this sort of error because the domain of the `Options` type is incredibly broad: it includes all objects with a `title` property that’s a `string` and *any other properties*, so long as those don’t include a `darkMode` property set to something other than `true` or `false`.

It’s easy to forget how expansive TypeScript types can be. Here are a few more values that are assignable to `Options`:

```
const o1: Options = document; // OK
const o2: Options = new HTMLAnchorElement; // OK
```

Both `document` and instances of `HTMLAnchorElement` have `title` properties that are strings, so these assignments are OK. `Options` is a broad type indeed!

Excess property checking tries to rein this in without compromising the fundamentally structural nature of the type system. It does this by disallowing unknown properties specifically on object literals. (It's sometimes called “strict object literal checking” for this reason.) Neither `document` nor `new HTMLAnchorElement` is an object literal, so they did not trigger the checks. But the `{title, darkmode}` object is, so it does:

```
const o: Options = { darkmode: true, title: 'Ski Free' };
// ~~~~~ 'darkmode' does not exist in type 'Options'...
```

This explains why using an intermediate variable without a type annotation makes the error go away:

```
const intermediate = { darkmode: true, title: 'Ski Free' };
const o: Options = intermediate; // OK
```

While the righthand side of the first line is an object literal, the righthand side of the second line (`intermediate`) is not, so excess property checking does not apply, and the error goes away.

Excess property checking does not happen when you use a type assertion:

```
const o = { darkmode: true, title: 'Ski Free' } as Options; // OK
```

This is a good reason to prefer declarations to assertions ([Item 9](#)).

If you don't want this sort of check, you can tell TypeScript to expect additional properties using an index signature:

```
interface Options {
  darkMode?: boolean;
  [otherOptions: string]: unknown;
}
const o: Options = { darkmode: true }; // OK
```

[Item 15](#) discusses when this is and is not an appropriate way to model your data.

A related check happens for “weak” types, which have only optional properties:

```
interface LineChartOptions {
  logscale?: boolean;
  invertedYAxis?: boolean;
  areaChart?: boolean;
}
const opts = { logScale: true };
const o: LineChartOptions = opts;
```



```
// ~ Type '{ logScale: boolean; }' has no properties in common
// with type 'LineChartOptions'
```

From a structural point of view, the `LineChartOptions` type should include almost all objects. For weak types like this, TypeScript adds another check to make sure that the value type and declared type have at least one property in common. Much like excess property checking, this is effective at catching typos and isn't strictly structural. But unlike excess property checking, it happens during all assignability checks involving weak types. Factoring out an intermediate variable doesn't bypass this check.

Excess property checking is an effective way of catching typos and other mistakes in property names that would otherwise be allowed by the structural typing system. It's particularly useful with types like `Options` that contain optional fields. But it is also very limited in scope: it only applies to object literals. Recognize this limitation and distinguish between excess property checking and ordinary type checking. This will help you build a mental model of both.

Factoring out a constant made an error go away here, but it can also introduce an error in other contexts. See [Item 26](#) for examples of this.

Things to Remember

- When you assign an object literal to a variable or pass it as an argument to a function, it undergoes excess property checking.
- Excess property checking is an effective way to find errors, but it is distinct from the usual structural assignability checks done by the TypeScript type checker. Conflating these processes will make it harder for you to build a mental model of assignability.
- Be aware of the limits of excess property checking: introducing an intermediate variable will remove these checks.

Item 12: Apply Types to Entire Function Expressions When Possible

JavaScript (and TypeScript) distinguishes a function *statement* and a function *expression*:

```
function rollDice1(sides: number): number { /* ... */ } // Statement
const rollDice2 = function(sides: number): number { /* ... */ }; // Expression
const rollDice3 = (sides: number): number => { /* ... */ }; // Also expression
```

An advantage of function expressions in TypeScript is that you can apply a type declaration to the entire function at once, rather than specifying the types of the parameters and return type individually:

```

type DiceRollFn = (sides: number) => number;
const rollDice: DiceRollFn = sides => { /* ... */ };

```

If you mouse over `sides` in your editor, you'll see that TypeScript knows its type is `number`. The function type doesn't provide much value in such a simple example, but the technique does open up a number of possibilities.

One is reducing repetition. If you wanted to write several functions for doing arithmetic on numbers, for instance, you could write them like this:

```

function add(a: number, b: number) { return a + b; }
function sub(a: number, b: number) { return a - b; }
function mul(a: number, b: number) { return a * b; }
function div(a: number, b: number) { return a / b; }

```

or consolidate the repeated function signatures with a single function type:

```

type BinaryFn = (a: number, b: number) => number;
const add: BinaryFn = (a, b) => a + b;
const sub: BinaryFn = (a, b) => a - b;
const mul: BinaryFn = (a, b) => a * b;
const div: BinaryFn = (a, b) => a / b;

```

This has fewer type annotations than before, and they're separated away from the function implementations. This makes the logic more apparent. You've also gained a check that the return type of all the function expressions is `number`.

Libraries often provide types for common function signatures. For example, ReactJS provides a `MouseEventHandler` type that you can apply to an entire function rather than specifying `MouseEvent` as a type for the function's parameter. If you're a library author, consider providing type declarations for common callbacks.

Another place you might want to apply a type to a function expression is to match the signature of some other function. In a web browser, for example, the `fetch` function issues an HTTP request for some resource:

```

const responseP = fetch('/quote?by=Mark+Twain'); // Type is Promise<Response>

```

You extract data from the response via `response.json()` or `response.text()`:

```

async function getQuote() {
  const response = await fetch('/quote?by=Mark+Twain');
  const quote = await response.json();
  return quote;
}
// {
//   "quote": "If you tell the truth, you don't have to remember anything.",
//   "source": "notebook",
//   "date": "1894"
// }

```

(See [Item 25](#) for more on Promises and `async/await`.)

There's a bug here: if the request for `/quote` fails, the response body is likely to contain an explanation like "404 Not Found." This isn't JSON, so `response.json()` will return a rejected Promise with a message about invalid JSON. This obscures the real error, which was a 404.

It's easy to forget that an error response with `fetch` does not result in a rejected Promise. Let's write a `checkedFetch` function to do the status check for us. The type declarations for `fetch` in `lib.dom.d.ts` look like this:

```
declare function fetch(
  input: RequestInfo, init?: RequestInit
): Promise<Response>;
```

So you can write `checkedFetch` like this:

```
async function checkedFetch(input: RequestInfo, init?: RequestInit) {
  const response = await fetch(input, init);
  if (!response.ok) {
    // Converted to a rejected Promise in an async function
    throw new Error('Request failed: ' + response.status);
  }
  return response;
}
```

This works, but it can be written more concisely:

```
const checkedFetch: typeof fetch = async (input, init) => {
  const response = await fetch(input, init);
  if (!response.ok) {
    throw new Error('Request failed: ' + response.status);
  }
  return response;
}
```

We've changed from a function statement to a function expression and applied a type (`typeof fetch`) to the entire function. This allows TypeScript to infer the types of the input and `init` parameters.

The type annotation also guarantees that the return type of `checkedFetch` will be the same as that of `fetch`. Had you written `return` instead of `throw`, for example, TypeScript would have caught the mistake:

```
const checkedFetch: typeof fetch = async (input, init) => {
  // ~~~~~ Type 'Promise<Response | HTTPError>'
  //           is not assignable to type 'Promise<Response>'
  //           Type 'Response | HTTPError' is not assignable
  //           to type 'Response'
  const response = await fetch(input, init);
  if (!response.ok) {
    return new Error('Request failed: ' + response.status);
  }
}
```

```
    return response;
}
```

The same mistake in the first example would likely have led to an error, but in the code that called `checkedFetch`, rather than in the implementation.

In addition to being more concise, typing this entire function expression instead of its parameters has given you better safety. When you're writing a function that has the same type signature as another one, or writing many functions with the same type signature, consider whether you can apply a type declaration to entire functions, rather than repeating types of parameters and return values.

Things to Remember

- Consider applying type annotations to entire function expressions, rather than to their parameters and return type.
- If you're writing the same type signature repeatedly, factor out a function type or look for an existing one. If you're a library author, provide types for common callbacks.
- Use `typeof fn` to match the signature of another function.

Item 13: Know the Differences Between type and interface

If you want to define a named type in TypeScript, you have two options. You can use a type, as shown here:

```
type TState = {
  name: string;
  capital: string;
}
```

or an interface:

```
interface IState {
  name: string;
  capital: string;
}
```

(You could also use a `class`, but that is a JavaScript runtime concept that also introduces a value. See [Item 8](#).)

Which should you use, type or interface? The line between these two options has become increasingly blurred over the years, to the point that in many situations you can use either. You should be aware of the distinctions that remain between type and interface and be consistent about which you use in which situation. But you should

also know how to write the same types using both, so that you'll be comfortable reading TypeScript that uses either.



The examples in this item prefix type names with `I` or `T` solely to indicate how they were defined. You should not do this in your code! Prefixing interface types with `I` is common in C#, and this convention made some inroads in the early days of TypeScript. But it is considered bad style today because it's unnecessary, adds little value, and is not consistently followed in the standard libraries.

First, the similarities: the `State` types are nearly indistinguishable from one another. If you define an `IState` or a `TState` value with an extra property, the errors you get are character-by-character identical:

```
const wyoming: TState = {
  name: 'Wyoming',
  capital: 'Cheyenne',
  population: 500_000
// ~~~~~ Type ... is not assignable to type 'TState'
//           Object literal may only specify known properties, and
//           'population' does not exist in type 'TState'
};
```

You can use an index signature with both interface and type:

```
type TDict = { [key: string]: string };
interface IDict {
  [key: string]: string;
}
```

You can also define function types with either:

```
type TFn = (x: number) => string;
interface IFn {
  (x: number): string;
}

const toStrT: TFn = x => '' + x; // OK
const toStrI: IFn = x => '' + x; // OK
```

The type alias looks more natural for this straightforward function type, but if the type has properties as well, then the declarations start to look more alike:

```
type TFnWithProperties = {
  (x: number): number;
  prop: string;
}
interface IFnWithProperties {
  (x: number): number;
  prop: string;
}
```

You can remember this syntax by reminding yourself that in JavaScript, functions are callable objects.

Both type aliases and interfaces can be generic:

```
type TPair<T> = {  
  first: T;  
  second: T;  
}  
interface IPair<T> {  
  first: T;  
  second: T;  
}
```

An interface can extend a type (with some caveats, explained momentarily), and a type can extend an interface:

```
interface IStateWithPop extends TState {  
  population: number;  
}  
type TStateWithPop = IState & { population: number; };
```

Again, these types are identical. The caveat is that an interface cannot extend a complex type like a union type. If you want to do that, you'll need to use type and &.

A class can implement either an interface or a simple type:

```
class StateT implements TState {  
  name: string = '';  
  capital: string = '';  
}  
class StateI implements IState {  
  name: string = '';  
  capital: string = '';  
}
```

Those are the similarities. What about the differences? You've seen one already—there are union types but no union interfaces:

```
type AorB = 'a' | 'b';
```

Extending union types can be useful. If you have separate types for Input and Output variables and a mapping from name to variable:

```
type Input = { /* ... */ };  
type Output = { /* ... */ };  
interface VariableMap {  
  [name: string]: Input | Output;  
}
```

then you might want a type that attaches the name to the variable. This would be:

```
type NamedVariable = (Input | Output) & { name: string };
```

This type cannot be expressed with `interface`. A `type` is, in general, more capable than an `interface`. It can be a union, and it can also take advantage of more advanced features like mapped or conditional types.

It can also more easily express tuple and array types:

```
type Pair = [number, number];
type StringList = string[];
type NamedNums = [string, ...number[]];
```

You can express something *like* a tuple using `interface`:

```
interface Tuple {
  0: number;
  1: number;
  length: 2;
}
const t: Tuple = [10, 20]; // OK
```

But this is awkward and drops all the tuple methods like `concat`. Better to use a `type`. For more on the problems of numeric indices, see [Item 16](#).

An `interface` does have some abilities that a `type` doesn't, however. One of these is that an `interface` can be *augmented*. Going back to the `State` example, you could have added a `population` field in another way:

```
interface IState {
  name: string;
  capital: string;
}
interface IState {
  population: number;
}
const wyoming: IState = {
  name: 'Wyoming',
  capital: 'Cheyenne',
  population: 500_000
}; // OK
```

This is known as “declaration merging,” and it’s quite surprising if you’ve never seen it before. This is primarily used with type declaration files ([Chapter 6](#)), and if you’re writing one, you should follow the norms and use `interface` to support it. The idea is that there may be gaps in your type declarations that users need to fill, and this is how they do it.

TypeScript uses merging to get different types for the different versions of JavaScript’s standard library. The `Array` interface, for example, is defined in `lib.es5.d.ts`. By default this is all you get. But if you add ES2015 to the `lib` entry of your `tsconfig.json`, TypeScript will also include `lib.es2015.d.ts`. This includes another `Array` interface with additional methods like `find` that were added in ES2015. They get added to the other

Array interface via merging. The net effect is that you get a single Array type with exactly the right methods.

Merging is supported in regular code as well as declarations, and you should be aware of the possibility. If it's essential that no one ever augment your type, then use `type`.

Returning to the question at the start of the item, should you use `type` or `interface`? For complex types, you have no choice: you need to use a type alias. But what about the simpler object types that can be represented either way? To answer this question, you should consider consistency and augmentation. Are you working in a codebase that consistently uses `interface`? Then stick with `interface`. Does it use `type`? Then use `type`.

For projects without an established style, you should think about augmentation. Are you publishing type declarations for an API? Then it might be helpful for your users to be able to be able to merge in new fields via an `interface` when the API changes. So use `interface`. But for a type that's used internally in your project, declaration merging is likely to be a mistake. So prefer `type`.

Things to Remember

- Understand the differences and similarities between `type` and `interface`.
- Know how to write the same types using either syntax.
- In deciding which to use in your project, consider the established style and whether augmentation might be beneficial.

Item 14: Use Type Operations and Generics to Avoid Repeating Yourself

This script prints the dimensions, surface areas, and volumes of a few cylinders:

```
console.log('Cylinder 1 x 1 ',
  'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 1 * 1,
  'Volume:', 3.14159 * 1 * 1 * 1);
console.log('Cylinder 1 x 2 ',
  'Surface area:', 6.283185 * 1 * 1 + 6.283185 * 2 * 1,
  'Volume:', 3.14159 * 1 * 2 * 1);
console.log('Cylinder 2 x 1 ',
  'Surface area:', 6.283185 * 2 * 1 + 6.283185 * 2 * 1,
  'Volume:', 3.14159 * 2 * 2 * 1);
```

Is this code uncomfortable to look at? It should be. It's extremely repetitive, as though the same line was copied and pasted, then modified. It repeats both values and con-

stants. This has allowed an error to creep in (did you spot it?). Much better would be to factor out some functions, a constant, and a loop:

```
const surfaceArea = (r, h) => 2 * Math.PI * r * (r + h);
const volume = (r, h) => Math.PI * r * r * h;
for (const [r, h] of [[1, 1], [1, 2], [2, 1]]) {
  console.log(
    `Cylinder ${r} x ${h}`,
    `Surface area: ${surfaceArea(r, h)}`,
    `Volume: ${volume(r, h)}`);
}
```

This is the DRY principle: don't repeat yourself. It's the closest thing to universal advice that you'll find in software development. Yet developers who assiduously avoid repetition in code may not think twice about it in types:

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate {
  firstName: string;
  lastName: string;
  birth: Date;
}
```

Duplication in types has many of the same problems as duplication in code. What if you decide to add an optional `middleName` field to `Person`? Now `Person` and `PersonWithBirthDate` have diverged.

One reason that duplication is more common in types is that the mechanisms for factoring out shared patterns are less familiar than they are with code: what's the type system equivalent of factoring out a helper function? By learning how to map between types, you can bring the benefits of DRY to your type definitions.

The simplest way to reduce repetition is by naming your types. Rather than writing a distance function this way:

```
function distance(a: {x: number, y: number}, b: {x: number, y: number}) {
  return Math.sqrt(Math.pow(a.x - b.x, 2) + Math.pow(a.y - b.y, 2));
}
```

create a name for the type and use that:

```
interface Point2D {
  x: number;
  y: number;
}
function distance(a: Point2D, b: Point2D) { /* ... */ }
```

This is the type system equivalent of factoring out a constant instead of writing it repeatedly. Duplicated types aren't always so easy to spot. Sometimes they can be obscured by syntax. If several functions share the same type signature, for instance:

```
function get(url: string, opts: Options): Promise<Response> { /* ... */ }
function post(url: string, opts: Options): Promise<Response> { /* ... */ }
```

Then you can factor out a named type for this signature:

```
type HTTPFunction = (url: string, opts: Options) => Promise<Response>;
const get: HTTPFunction = (url, opts) => { /* ... */ };
const post: HTTPFunction = (url, opts) => { /* ... */ };
```

For more on this, see [Item 12](#).

What about the `Person/PersonWithBirthDate` example? You can eliminate the repetition by making one interface extend the other:

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate extends Person {
  birth: Date;
}
```

Now you only need to write the additional fields. If the two interfaces share a subset of their fields, then you can factor out a base class with just these common fields. Continuing the analogy with code duplication, this is akin to writing π and 2π instead of 3.141593 and 6.283185.

You can also use the intersection operator (`&`) to extend an existing type, though this is less common:

```
type PersonWithBirthDate = Person & { birth: Date };
```

This technique is most useful when you want to add some additional properties to a union type (which you cannot extend). For more on this, see [Item 13](#).

You can also go the other direction. What if you have a type, `State`, which represents the state of an entire application, and another, `TopNavState`, which represents just a part?

```
interface State {
  userId: string;
  pageTitle: string;
  recentFiles: string[];
  pageContents: string;
}

interface TopNavState {
  userId: string;
  pageTitle: string;
}
```

```
    recentFiles: string[];
}
```

Rather than building up State by extending TopNavState, you'd like to define TopNavState as a subset of the fields in State. This way you can keep a single interface defining the state for the entire app.

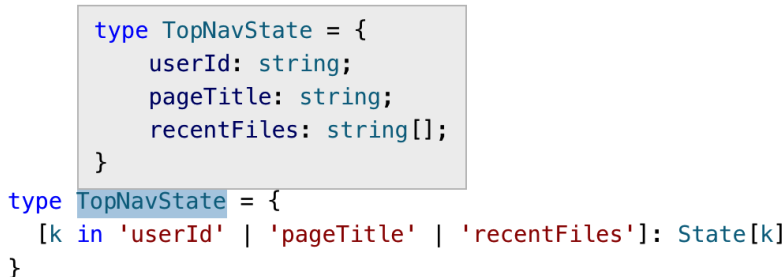
You can remove duplication in the types of the properties by indexing into State:

```
type TopNavState = {
  userId: State['userId'];
  pageTitle: State['pageTitle'];
  recentFiles: State['recentFiles'];
};
```

While it's longer, this *is* progress: a change in the type of pageTitle in State will get reflected in TopNavState. But it's still repetitive. You can do better with a *mapped type*:

```
type TopNavState = {
  [k in 'userId' | 'pageTitle' | 'recentFiles']: State[k]
};
```

Mousing over TopNavState shows that this definition is, in fact, exactly the same as the previous one (see [Figure 2-10](#)).



```
type TopNavState = {
  userId: string;
  pageTitle: string;
  recentFiles: string[];
}

type TopNavState = {
  [k in 'userId' | 'pageTitle' | 'recentFiles']: State[k]
}
```

Figure 2-10. Showing the expanded version of a mapped type in your text editor. This is the same as the initial definition, but with less duplication.

Mapped types are the type system equivalent of looping over the fields in an array. This particular pattern is so common that it's part of the standard library, where it's called `Pick`:

```
type Pick<T, K> = { [k in K]: T[k] };
```

(This definition isn't *quite* complete, as you will see.) You use it like this:

```
type TopNavState = Pick<State, 'userId' | 'pageTitle' | 'recentFiles'>;
```

Pick is an example of a *generic type*. Continuing the analogy to removing code duplication, using Pick is the equivalent of calling a function. Pick takes two types, T and K, and returns a third, much as a function might take two values and return a third.

Another form of duplication can arise with tagged unions. What if you want a type for just the tag?

```
interface SaveAction {
  type: 'save';
  // ...
}
interface LoadAction {
  type: 'load';
  // ...
}
type Action = SaveAction | LoadAction;
type ActionType = 'save' | 'load'; // Repeated types!
```

You can define ActionType without repeating yourself by indexing into the Action union:

```
type ActionType = Action['type']; // Type is "save" | "load"
```

As you add more types to the Action union, ActionType will incorporate them automatically. This type is distinct from what you'd get using Pick, which would give you an interface with a type property:

```
type ActionRec = Pick<Action, 'type'>; // {type: "save" | "load"}
```

If you're defining a class which can be initialized and later updated, the type for the parameter to the update method will optionally include most of the same parameters as the constructor:

```
interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
interface OptionsUpdate {
  width?: number;
  height?: number;
  color?: string;
  label?: string;
}
class UIWidget {
  constructor(init: Options) { /* ... */ }
  update(options: OptionsUpdate) { /* ... */ }
}
```

You can construct OptionsUpdate from Options using a mapped type and keyof:

```
type OptionsUpdate = {[k in keyof Options]?: Options[k]};
```

`keyof` takes a type and gives you a union of the types of its keys:

```
type OptionsKeys = keyof Options;
// Type is "width" | "height" | "color" | "label"
```

The mapped type (`[k in keyof Options]`) iterates over these and looks up the corresponding value type in `Options`. The `?` makes each property optional. This pattern is also extremely common and is enshrined in the standard library as `Partial`:

```
class UIWidget {
  constructor(init: Options) { /* ... */ }
  update(options: Partial<Options>) { /* ... */ }
}
```

You may also find yourself wanting to define a type that matches the shape of a *value*:

```
const INIT_OPTIONS = {
  width: 640,
  height: 480,
  color: '#00FF00',
  label: 'VGA',
};
interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
```

You can do so with `typeof`:

```
type Options = typeof INIT_OPTIONS;
```

This intentionally evokes JavaScript's runtime `typeof` operator, but it operates at the level of TypeScript types and is much more precise. For more on `typeof`, see [Item 8](#). Be careful about deriving types from values, however. It's usually better to define types first and declare that values are assignable to them. This makes your types more explicit and less subject to the vagaries of widening ([Item 21](#)).

Similarly, you may want to create a named type for the inferred return value of a function or method:

```
function getUserInfo(userId: string) {
  // ...
  return {
    userId,
    name,
    age,
    height,
    weight,
    favoriteColor,
  };
};
```

```

}
// Return type inferred as { userId: string; name: string; age: number, ... }

```

Doing this directly requires conditional types (see [Item 50](#)). But, as we’ve seen before, the standard library defines generic types for common patterns like this one. In this case the `ReturnType` generic does exactly what you want:

```

type UserInfo = ReturnType<typeof getUserInfo>;

```

Note that `ReturnType` operates on `typeof getUserInfo`, the function’s *type*, rather than `getUserInfo`, the function’s *value*. As with `typeof`, use this technique judiciously. Don’t get mixed up about your source of truth.

Generic types are the equivalent of functions for types. And functions are the key to DRY for logic. So it should come as no surprise that generics are the key to DRY for types. But there’s a missing piece to this analogy. You use the type system to constrain the values you can map with a function: you add numbers, not objects; you find the area of shapes, not database records. How do you constrain the parameters in a generic type?

You do so with `extends`. You can declare that any generic parameter extends a type. For example:

```

interface Name {
  first: string;
  last: string;
}
type DancingDuo<T extends Name> = [T, T];

const couple1: DancingDuo<Name> = [
  {first: 'Fred', last: 'Astaire'},
  {first: 'Ginger', last: 'Rogers'}
]; // OK

const couple2: DancingDuo<{first: string}> = [
  // ~~~~~
  // Property 'last' is missing in type
  // '{ first: string; }' but required in type 'Name'
  {first: 'Sonny'},
  {first: 'Cher'}
];

```

`{first: string}` does not extend `Name`, hence the error.



At the moment, TypeScript always requires you to write out the generic parameter in a declaration. Writing `DancingDuo` instead of `DancingDuo<Name>` won't cut it. If you want TypeScript to infer the type of the generic parameter, you can use a carefully typed identity function:

```
const dancingDuo = <T extends Name>(x: DancingDuo<T>) => x;
const couple1 = dancingDuo([
  {first: 'Fred', last: 'Astaire'},
  {first: 'Ginger', last: 'Rogers'}
]);
const couple2 = dancingDuo([
  {first: 'Bono'},
  // ~~~~~
  {first: 'Prince'}
  // ~~~~~
  //      Property 'last' is missing in type
  //      '{ first: string; }' but required in type 'Name'
]);
```

For a particularly useful variation on this, see [inferringPick](#) in [Item 26](#).

You can use `extends` to complete the definition of `Pick` from earlier. If you run the original version through the type checker, you get an error:

```
type Pick<T, K> = {
  [k in K]: T[k]
  // ~ Type 'K' is not assignable to type 'string | number | symbol'
};
```

`K` is unconstrained in this type and is clearly too broad: it needs to be something that can be used as an index, namely, `string | number | symbol`. But you can get narrower than that—`K` should really be some subset of the keys of `T`, namely, `keyof T`:

```
type Pick<T, K extends keyof T> = {
  [k in K]: T[k]
}; // OK
```

Thinking of types as sets of values ([Item 7](#)), it helps to read “`extends`” as “subset of” here.

As you work with increasingly abstract types, try not to lose sight of the goal: accepting valid programs and rejecting invalid ones. In this case, the upshot of the constraint is that passing `Pick` the wrong key will produce an error:

```
type FirstLast = Pick<Name, 'first' | 'last'>; // OK
type FirstMiddle = Pick<Name, 'first' | 'middle'>;
// ~~~~~
// Type '"middle"' is not assignable
// to type '"first" | "last"'
```

Repetition and copy/paste coding are just as bad in type space as they are in value space. The constructs you use to avoid repetition in type space may be less familiar than those used for program logic, but they are worth the effort to learn. Don't repeat yourself!

Things to Remember

- The DRY (don't repeat yourself) principle applies to types as much as it applies to logic.
- Name types rather than repeating them. Use `extends` to avoid repeating fields in interfaces.
- Build an understanding of the tools provided by TypeScript to map between types. These include `keyof`, `typeof`, indexing, and mapped types.
- Generic types are the equivalent of functions for types. Use them to map between types instead of repeating types. Use `extends` to constrain generic types.
- Familiarize yourself with generic types defined in the standard library such as `Pick`, `Partial`, and `ReturnType`.

Item 15: Use Index Signatures for Dynamic Data

One of the best features of JavaScript is its convenient syntax for creating objects:

```
const rocket = {  
  name: 'Falcon 9',  
  variant: 'Block 5',  
  thrust: '7,607 kN',  
};
```

Objects in JavaScript map string keys to values of any type. TypeScript lets you represent flexible mappings like this by specifying an *index signature* on the type:

```
type Rocket = {[property: string]: string};  
const rocket: Rocket = {  
  name: 'Falcon 9',  
  variant: 'v1.0',  
  thrust: '4,940 kN',  
}; // OK
```

The `[property: string]: string` is the index signature. It specifies three things:

A name for the keys

This is purely for documentation; it is not used by the type checker in any way.

A type for the key

This needs to be some combination of string, number, or symbol, but generally you just want to use string (see [Item 16](#)).

A type for the values

This can be anything.

While this does type check, it has a few downsides:

- It allows any keys, including incorrect ones. Had you written `Name` instead of `name`, it would have still been a valid `Rocket` type.
- It doesn't require any specific keys to be present. `{}` is also a valid `Rocket`.
- It cannot have distinct types for different keys. For example, `thrust` should probably be a number, not a string.
- TypeScript's language services can't help you with types like this. As you're typing `name:`, there's no autocomplete because the key could be anything.

In short, index signatures are not very precise. There are almost always better alternatives to them. In this case, `Rocket` should clearly be an interface:

```
interface Rocket {  
  name: string;  
  variant: string;  
  thrust_kN: number;  
}  
  
const falconHeavy: Rocket = {  
  name: 'Falcon Heavy',  
  variant: 'v1',  
  thrust_kN: 15_200  
};
```

Now `thrust_kN` is a number and TypeScript will check for the presence of all required fields. All the great language services that TypeScript provides are available: autocomplete, jump to definition, rename—and they all work.

What should you use index signatures for? The canonical case is truly dynamic data. This might come from a CSV file, for instance, where you have a header row and want to represent data rows as objects mapping column names to values:

```
function parseCSV(input: string): {[columnName: string]: string}[] {  
  const lines = input.split('\n');  
  const [header, ...rows] = lines;  
  return rows.map(rowStr => {  
    const row: {[columnName: string]: string} = {};  
    rowStr.split(',').forEach((cell, i) => {  
      row[header[i]] = cell;  
    });  
    return row;  
  });  
}
```

```
    });
  }
}
```

There's no way to know in advance what the column names are in such a general setting. So an index signature is appropriate. If the user of `parseCSV` knows more about what the columns are in a particular context, they may want to use an assertion to get a more specific type:

```
interface ProductRow {
  productId: string;
  name: string;
  price: string;
}

declare let csvData: string;
const products = parseCSV(csvData) as unknown as ProductRow[];
```

Of course, there's no guarantee that the columns at runtime will actually match your expectation. If this is something you're concerned about, you can add `undefined` to the value type:

```
function safeParseCSV(
  input: string
): {[columnName: string]: string | undefined}[] {
  return parseCSV(input);
}
```

Now every access requires a check:

```
const rows = parseCSV(csvData);
const prices: {[product: string]: number} = {};
for (const row of rows) {
  prices[row.productId] = Number(row.price);
}

const safeRows = safeParseCSV(csvData);
for (const row of safeRows) {
  prices[row.productId] = Number(row.price);
  // ~~~~~ Type 'undefined' cannot be used as an index type
}
```

Of course, this may make the type less convenient to work with. Use your judgment.

If your type has a limited set of possible fields, don't model this with an index signature. For instance, if you know your data will have keys like A, B, C, D, but you don't know how many of them there will be, you could model the type either with optional fields or a union:

```
interface Row1 { [column: string]: number } // Too broad
interface Row2 { a: number; b?: number; c?: number; d?: number } // Better
type Row3 =
  | { a: number; }
  | { a: number; b: number; }
```

```
| { a: number; b: number; c: number; }
| { a: number; b: number; c: number; d: number };
```

The last form is the most precise, but it may be less convenient to work with.

If the problem with using an index signature is that `string` is too broad, then there are a few alternatives.

One is using `Record`. This is a generic type that gives you more flexibility in the key type. In particular, you can pass in subsets of `string`:

```
type Vec3D = Record<'x' | 'y' | 'z', number>;
// Type Vec3D = {
//   x: number;
//   y: number;
//   z: number;
// }
```

Another is using a mapped type. This gives you the possibility of using different types for different keys:

```
type Vec3D = {[k in 'x' | 'y' | 'z']: number};
// Same as above
type ABC = {[k in 'a' | 'b' | 'c']: k extends 'b' ? string : number};
// Type ABC = {
//   a: number;
//   b: string;
//   c: number;
// }
```

Things to Remember

- Use index signatures when the properties of an object cannot be known until runtime—for example, if you’re loading them from a CSV file.
- Consider adding `undefined` to the value type of an index signature for safer access.
- Prefer more precise types to index signatures when possible: interfaces, Records, or mapped types.

Item 16: Prefer Arrays, Tuples, and ArrayLike to number Index Signatures

JavaScript is a famously quirky language. Some of the most notorious quirks involve implicit type coercions:

```
> "0" == 0
true
```

but these can usually be avoided by using `===` and `!==` instead of their more coercive cousins.

JavaScript's object model also has its quirks, and these are more important to understand because some of them are modeled by TypeScript's type system. You've already seen one such quirk in [Item 10](#), which discussed object wrapper types. This item discusses another.

What is an object? In JavaScript it's a collection of key/value pairs. The keys are usually strings (in ES2015 and later they can also be symbols). The values can be anything.

This is more restrictive than what you find in many other languages. JavaScript does not have a notion of “hashable” objects like you find in Python or Java. If you try to use a more complex object as a key, it is converted into a string by calling its `toString` method:

```
> x = {}  
{}  
> x[[1, 2, 3]] = 2  
2  
> x  
{ '1,2,3': 1 }
```

In particular, *numbers* cannot be used as keys. If you try to use a number as a property name, the JavaScript runtime will convert it to a string:

```
> { 1: 2, 3: 4 }  
{ '1': 2, '3': 4 }
```

So what are arrays, then? They are certainly objects:

```
> typeof []  
'object'
```

And yet it's quite normal to use numeric indices with them:

```
> x = [1, 2, 3]  
[ 1, 2, 3 ]  
> x[0]  
1
```

Are these being converted into strings? In one of the oddest quirks of all, the answer is “yes.” You can also access the elements of an array using string keys:

```
> x['1']  
2
```

If you use `Object.keys` to list the keys of an array, you get strings back:

```
> Object.keys(x)  
[ '0', '1', '2' ]
```

TypeScript attempts to bring some sanity to this by allowing numeric keys and distinguishing between these and strings. If you dig into the type declarations for `Array` (Item 6), you'll find this in `lib.es5.d.ts`:

```
interface Array<T> {  
    // ...  
    [n: number]: T;  
}
```

This is purely a fiction—string keys are accepted at runtime as the ECMAScript standard dictates that they must—but it is a helpful one that can catch mistakes:

```
const xs = [1, 2, 3];  
const x0 = xs[0]; // OK  
const x1 = xs['1'];  
    // ~~~ Element implicitly has an 'any' type  
    //      because index expression is not of type 'number'  
  
function get<T>(array: T[], k: string): T {  
    return array[k];  
    // ~ Element implicitly has an 'any' type  
    //      because index expression is not of type 'number'  
}
```

While this fiction is helpful, it's important to remember that it is just a fiction. Like all aspects of TypeScript's type system, it is erased at runtime (Item 3). This means that constructs like `Object.keys` still return strings:

```
const keys = Object.keys(xs); // Type is string[]  
for (const key in xs) {  
    key; // Type is string  
    const x = xs[key]; // Type is number  
}
```

That this last access works is somewhat surprising since `string` is not assignable to `number`. It's best thought of as a pragmatic concession to this style of iterating over arrays, which is common in JavaScript. That's not to say that this is a good way to loop over an array. If you don't care about the index, you can use `for-of`:

```
for (const x of xs) {  
    x; // Type is number  
}
```

If you do care about the index, you can use `Array.prototype.forEach`, which gives it to you as a number:

```
xs.forEach((x, i) => {  
    i; // Type is number  
    x; // Type is number  
});
```

If you need to break out of the loop early, you're best off using a C-style `for(;;)` loop:

```
for (let i = 0; i < xs.length; i++) {
  const x = xs[i];
  if (x < 0) break;
}
```

If the types don't convince you, perhaps the performance will: in most browsers and JavaScript engines, for-in loops over arrays are several orders of magnitude slower than for-of or a C-style for loop.

The general pattern here is that a `number` index signature means that what you put in has to be a number (with the notable exception of for-in loops), but what you get out is a `string`.

If this sounds confusing, it's because it is! As a general rule, there's not much reason to use `number` as the index signature of a type rather than `string`. If you want to specify something that will be indexed using numbers, you probably want to use an `Array` or `tuple` type instead. Using `number` as an index type can create the misconception that numeric properties are a thing in JavaScript, either for yourself or for readers of your code.

If you object to accepting an `Array` type because they have many other properties (from their prototype) that you might not use, such as `push` and `concat`, then that's good—you're thinking structurally! (If you need a refresher on this, refer to [Item 4](#).) If you truly want to accept tuples of any length or any array-like construct, TypeScript has an `ArrayLike` type you can use:

```
function checkedAccess<T>(xs: ArrayLike<T>, i: number): T {
  if (i < xs.length) {
    return xs[i];
  }
  throw new Error(`Attempt to access ${i} which is past end of array.`)
}
```

This has just a `length` and numeric index signature. In the rare cases that this is what you want, you should use it instead. But remember that the keys are still really strings!

```
const tupleLike: ArrayLike<string> = {
  '0': 'A',
  '1': 'B',
  length: 2,
}; // OK
```

Things to Remember

- Understand that arrays are objects, so their keys are strings, not numbers. `number` as an index signature is a purely TypeScript construct which is designed to help catch bugs.

- Prefer Array, tuple, or ArrayLike types to using number in an index signature yourself.

Item 17: Use readonly to Avoid Errors Associated with Mutation

Here's some code to print the triangular numbers (1, 1+2, 1+2+3, etc.):

```
function printTriangles(n: number) {
  const nums = [];
  for (let i = 0; i < n; i++) {
    nums.push(i);
    console.log(arraySum(nums));
  }
}
```

This code looks straightforward. But here's what happens when you run it:

```
> printTriangles(5)
0
1
2
3
4
```

The problem is that you've made an assumption about `arraySum`, namely, that it doesn't modify `nums`. But here's my implementation:

```
function arraySum(arr: number[]) {
  let sum = 0, num;
  while ((num = arr.pop()) !== undefined) {
    sum += num;
  }
  return sum;
}
```

This function does calculate the sum of the numbers in the array. But it also has the side effect of emptying the array! TypeScript is fine with this, because JavaScript arrays are mutable.

It would be nice to have some assurances that `arraySum` does not modify the array. This is what the `readonly` type modifier does:

```
function arraySum(arr: readonly number[]) {
  let sum = 0, num;
  while ((num = arr.pop()) !== undefined) {
    // ~~~ 'pop' does not exist on type 'readonly number[]'
    sum += num;
  }
}
```

```
    return sum;
}
```

This error message is worth digging into. `readonly number[]` is a *type*, and it is distinct from `number[]` in a few ways:

- You can read from its elements, but you can't write to them.
- You can read its `length`, but you can't set it (which would mutate the array).
- You can't call `pop` or other methods that mutate the array.

Because `number[]` is strictly more capable than `readonly number[]`, it follows that `number[]` is a subtype of `readonly number[]`. (It's easy to get this backwards—remember [Item 7](#)!) So you can assign a mutable array to a `readonly` array, but not vice versa:

```
const a: number[] = [1, 2, 3];
const b: readonly number[] = a;
const c: number[] = b;
// ~ Type 'readonly number[]' is 'readonly' and cannot be
//   assigned to the mutable type 'number[]'
```

This makes sense: the `readonly` modifier wouldn't be much use if you could get rid of it without even a type assertion.

When you declare a parameter `readonly`, a few things happen:

- TypeScript checks that the parameter isn't mutated in the function body.
- Callers are assured that your function doesn't mutate the parameter.
- Callers may pass your function a `readonly` array.

There is often an assumption in JavaScript (and TypeScript) that functions don't mutate their parameters unless explicitly noted. But as we'll see time and again in this book (particularly [Items 30](#) and [31](#)), these sorts of implicit understandings can lead to trouble with type checking. Better to make them explicit, both for human readers and for `tsc`.

The fix for `arraySum` is simple: don't mutate the array!

```
function arraySum(arr: readonly number[]) {
    let sum = 0;
    for (const num of arr) {
        sum += num;
    }
    return sum;
}
```

Now `printTriangles` does what you expect:


```
> printTriangles(5)
0
1
3
6
10
```

If your function does not mutate its parameters, then you should declare them `readonly`. There's relatively little downside: users will be able to call them with a broader set of types (Item 29), and inadvertent mutations will be caught.

One downside is that you may need to call functions that haven't marked their parameters `readonly`. If these don't mutate their parameters and are in your control, make them `readonly`! `readonly` tends to be contagious: once you mark one function with `readonly`, you'll also need to mark all the functions that it calls. This is a good thing since it leads to clearer contracts and better type safety. But if you're calling a function in another library, you may not be able to change its type declarations, and you may have to resort to a type assertion (`param as number[]`).

`readonly` can also be used to catch a whole class of mutation errors involving local variables. Imagine you're writing a tool to process a novel. You get a sequence of lines and would like to collect them into paragraphs, which are separated by blanks:

```
Frankenstein; or, The Modern Prometheus
by Mary Shelley
```

```
You will rejoice to hear that no disaster has accompanied the commencement
of an enterprise which you have regarded with such evil forebodings. I arrived
here yesterday, and my first task is to assure my dear sister of my welfare and
increasing confidence in the success of my undertaking.
```

```
I am already far north of London, and as I walk in the streets of Petersburg,
I feel a cold northern breeze play upon my cheeks, which braces my nerves and
fills me with delight.
```

Here's an attempt:¹

```
function parseTaggedText(lines: string[]): string[][] {
  const paragraphs: string[][] = [];
  const currPara: string[] = [];

  const addParagraph = () => {
    if (currPara.length) {
      paragraphs.push(currPara);
      currPara.length = 0; // Clear the lines
    }
  };
```

¹ In practice you might just write `lines.join('\n').split(/\n\n+/)`, but bear with me.

```

    for (const line of lines) {
        if (!line) {
            addParagraph();
        } else {
            currPara.push(line);
        }
    }
    addParagraph();
    return paragraphs;
}

```

When you run this on the example at the beginning of the item, here's what you get:

```
[ [], [], [] ]
```

Well that went horribly wrong!

The problem with this code is a toxic combination of aliasing ([Item 24](#)) and mutation. The aliasing happens on this line:

```
paragraphs.push(currPara);
```

Rather than pushing the contents of `currPara`, this pushes a reference to the array. When you push a new value to `currPara` or clear it, this change is also reflected in the entries in `paragraphs` because they point to the same object.

In other words, the net effect of this code:

```

paragraphs.push(currPara);
currPara.length = 0; // Clear lines

```

is that you push a new paragraph onto `paragraphs` and then immediately clear it.

The problem is that setting `currPara.length` and calling `currPara.push` both mutate the `currPara` array. You can disallow this behavior by declaring it to be `readonly`. This immediately surfaces a few errors in the implementation:

```

function parseTaggedText(lines: string[]): string[][] {
    const currPara: readonly string[] = [];
    const paragraphs: string[][] = [];

    const addParagraph = () => {
        if (currPara.length) {
            paragraphs.push(
                currPara
                // ~~~~~ Type 'readonly string[]' is 'readonly' and
                // cannot be assigned to the mutable type 'string[]'
            );
            currPara.length = 0; // Clear lines
            // ~~~~~ Cannot assign to 'length' because it is a read-only
            // property
        }
    };
};

```

```

    for (const line of lines) {
      if (!line) {
        addParagraph();
      } else {
        currPara.push(line);
        // ~~~~ Property 'push' does not exist on type 'readonly string[]'
      }
    }
    addParagraph();
    return paragraphs;
  }
}

```

You can fix two of the errors by declaring `currPara` with `let` and using nonmutating methods:

```

let currPara: readonly string[] = [];
// ...
currPara = []; // Clear lines
// ...
currPara = currPara.concat([line]);

```

Unlike `push`, `concat` returns a new array, leaving the original unmodified. By changing the declaration from `const` to `let` and adding `readonly`, you've traded one sort of mutability for another. The `currPara` variable is now free to change which array it points to, but those arrays themselves are not allowed to change.

This leaves the error about paragraphs. You have three options for fixing this.

First, you could make a copy of `currPara`:

```

paragraphs.push([...currPara]);

```

This fixes the error because, while `currPara` remains `readonly`, you're free to mutate the copy however you like.

Second, you could change `paragraphs` (and the return type of the function) to be an array of `readonly string[]`:

```

const paragraphs: (readonly string[])[] = [];

```

(The grouping is relevant here: `readonly string[][]` would be a `readonly` array of mutable arrays, rather than a mutable array of `readonly` arrays.)

This works, but it seems a bit rude to users of `parseTaggedText`. Why do you care what they do with the paragraphs after the function returns?

Third, you could use an assertion to remove the `readonly`-ness of the array:

```

paragraphs.push(currPara as string[]);

```

Since you're assigning `currPara` to a new array in the very next statement, this doesn't seem like the most offensive assertion.

An important caveat to `readonly` is that it is *shallow*. You saw this with `readonly string[][]` earlier. If you have a `readonly` array of objects, the objects themselves are not `readonly`:

```
const dates: readonly Date[] = [new Date()];
dates.push(new Date());
// ~~~~ Property 'push' does not exist on type 'readonly Date[]'
dates[0].setFullYear(2037); // OK
```

Similar considerations apply to `readonly`'s cousin for objects, the `Readonly` generic:

```
interface Outer {
  inner: {
    x: number;
  }
}
const o: Readonly<Outer> = { inner: { x: 0 } };
o.inner = { x: 1 };
// ~~~~ Cannot assign to 'inner' because it is a read-only property
o.inner.x = 1; // OK
```

You can create a type alias and then inspect it in your editor to see exactly what's happening:

```
type T = Readonly<Outer>;
// Type T = {
//   readonly inner: {
//     x: number;
//   };
// }
```

The important thing to note is the `readonly` modifier on `inner` but not on `x`. There is no built-in support for deep `readonly` types at the time of this writing, but it is possible to create a generic to do this. Getting this right is tricky, so I recommend using a library rather than rolling your own. The `DeepReadonly` generic in `ts-essentials` is one implementation.

You can also write `readonly` on an index signature. This has the effect of preventing writes but allowing reads:

```
let obj: {readonly [k: string]: number} = {};
// Or Readonly<{[k: string]: number}>
obj.hi = 45;
// ~~ Index signature in type ... only permits reading
obj = {...obj, hi: 12}; // OK
obj = {...obj, bye: 34}; // OK
```

This can prevent issues with aliasing and mutation involving objects rather than arrays.

Things to Remember

- If your function does not modify its parameters then declare them `readonly`. This makes its contract clearer and prevents inadvertent mutations in its implementation.
- Use `readonly` to prevent errors with mutation and to find the places in your code where mutations occur.
- Understand the difference between `const` and `readonly`.
- Understand that `readonly` is shallow.

Item 18: Use Mapped Types to Keep Values in Sync

Suppose you're writing a UI component for drawing scatter plots. It has a few different types of properties that control its display and behavior:

```
interface ScatterProps {  
  // The data  
  xs: number[];  
  ys: number[];  
  
  // Display  
  xRange: [number, number];  
  yRange: [number, number];  
  color: string;  
  
  // Events  
  onClick: (x: number, y: number, index: number) => void;  
}
```

To avoid unnecessary work, you'd like to redraw the chart only when you need to. Changing data or display properties will require a redraw, but changing the event handler will not. This sort of optimization is common in React components, where an event handler Prop might be set to a new arrow function on every render.²

Here's one way you might implement this optimization:

```
function shouldUpdate(  
  oldProps: ScatterProps,  
  newProps: ScatterProps  
) {  
  let k: keyof ScatterProps;  
  for (k in oldProps) {  
    if (oldProps[k] !== newProps[k]) {
```

² React's `useCallback` hook is another technique to avoid creating new functions on every render.

```

        if (k !== 'onClick') return true;
    }
}
return false;
}

```

(See [Item 54](#) for an explanation of the `keyof` declaration in this loop.)

What happens when you or a coworker add a new property? The `shouldUpdate` function will redraw the chart whenever it changes. You might call this the conservative or “fail closed” approach. The upside is that the chart will always look right. The downside is that it might be drawn too often.

A “fail open” approach might look like this:

```

function shouldUpdate(
  oldProps: ScatterProps,
  newProps: ScatterProps
) {
  return (
    oldProps.xs !== newProps.xs ||
    oldProps.ys !== newProps.ys ||
    oldProps.xRange !== newProps.xRange ||
    oldProps.yRange !== newProps.yRange ||
    oldProps.color !== newProps.color
    // (no check for onClick)
  );
}

```

With this approach there won’t be any unnecessary redraws, but there might be some *necessary* draws that get dropped. This violates the “first, do no harm” principle of optimization and so is less common.

Neither approach is ideal. What you’d really like is to force your coworker or future self to make a decision when adding the new property. You might try adding a comment:

```

interface ScatterProps {
  xs: number[];
  ys: number[];
  // ...
  onClick: (x: number, y: number, index: number) => void;

  // Note: if you add a property here, update shouldUpdate!
}

```

But do you really expect this to work? It would be better if the type checker could enforce this for you.

If you set it up the right way, it can. The key is to use a mapped type and an object:

```

const REQUIRES_UPDATE: {[k in keyof ScatterProps]: boolean} = {
  xs: true,

```

```

    ys: true,
    xRange: true,
    yRange: true,
    color: true,
    onClick: false,
  };

  function shouldUpdate(
    oldProps: ScatterProps,
    newProps: ScatterProps
  ) {
    let k: keyof ScatterProps;
    for (k in oldProps) {
      if (oldProps[k] !== newProps[k] && REQUIRES_UPDATE[k]) {
        return true;
      }
    }
    return false;
  }
}

```

The `[k in keyof ScatterProps]` tells the type checker that `REQUIRES_UPDATE` should have all the same properties as `ScatterProps`. If future you adds a new property to `ScatterProps`:

```

interface ScatterProps {
  // ...
  onDoubleClick: () => void;
}

```

Then this will produce an error in the definition of `REQUIRES_UPDATE`:

```

const REQUIRES_UPDATE: {[k in keyof ScatterProps]: boolean} = {
  // ~~~~~~ Property 'onDoubleClick' is missing in type
  // ...
};

```

This will certainly force the issue! Deleting or renaming a property will cause a similar error.

It's important that we used an object with boolean values here. Had we used an array:

```

const PROPS_REQUIRING_UPDATE: (keyof ScatterProps)[] = [
  'xs',
  'ys',
  // ...
];

```

then we would have been forced into the same fail open/fail closed choice.

Mapped types are ideal if you want one object to have exactly the same properties as another. As in this example, you can use this to make TypeScript enforce constraints on your code.

Things to Remember

- Use mapped types to keep related values and types synchronized.
- Consider using mapped types to force choices when adding new properties to an interface.