Chapter 3. All About Types

In the last chapter I introduced the idea of type systems, but I never defined what the *type* in type system really means.

TYPE

A set of values and the things you can do with them.

If that sounds confusing, let me give a few familiar examples:

- The `boolean` type is the set of all booleans (there are just two: `true` and `false`) and the operations you can perform on them (like `||`, `&&`, and `!`).
- The `number` type is the set of all numbers and the operations you can perform on them (like `+`, `-`, *, `/`, `%`, `||`, `&&`, and `?`), including the methods you can call on them like `.toFixed`, `.toPrecision`, `.toString`, and so on.
- The `string` type is the set of all strings and the operations you can perform on them (like `+`, `||`, and `&&`), including the methods you can call on them like `.concat` and `.toUpperCase`.

When you see that something is of type `T`, not only do you know that it's a `T`, but you also know *exactly what you can do* with that `T` (and what you can't). Remember, the whole point is to use the typechecker to stop you from doing invalid things. And the way the typechecker knows what's valid and what's not is by looking at the types you're using and how you're using them.

In this chapter we'll take a tour of the types available in TypeScript and cover the basics of what you can do with each of them. Figure 3-1 gives an overview.

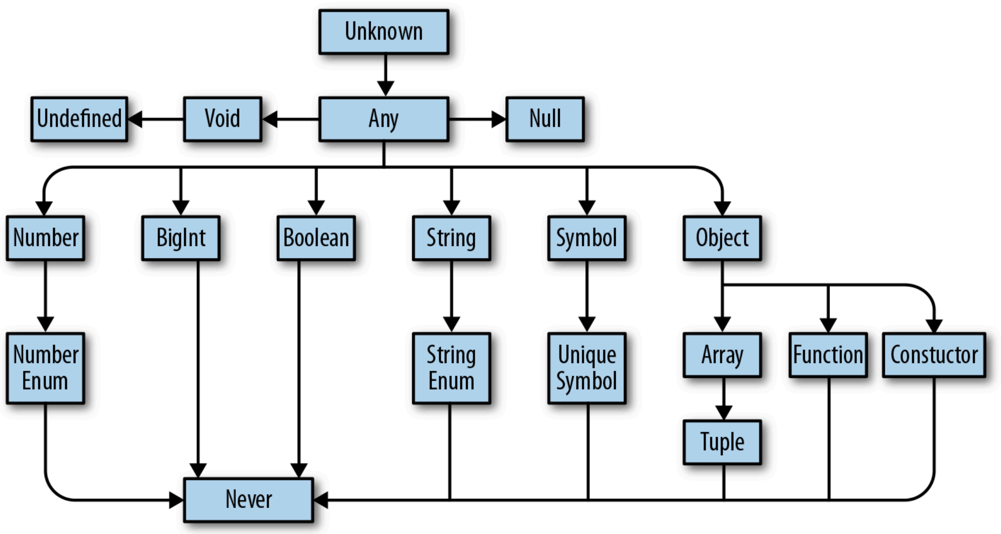

Figure 3-1. TypeScript's type hierarchy

Talking About Types

When programmers talk about types, they share a precise, common vocabulary to describe what they mean. We're going to use this vocabulary throughout this book.

Say you have a function that takes some value and returns that value multiplied by itself:

```
function squareOf(n) {
  return n * n
}
squareOf(2)      // evaluates to 4
squareOf('z')    // evaluates to NaN
```

Clearly, this function will only work for numbers—if you pass anything besides a number to `squareOf`, the result will be invalid. So what we do is explicitly *annotate* the parameter's type:

```
function squareOf(n: number) {
  return n * n
}
squareOf(2)      // evaluates to 4
squareOf('z')    // Error TS2345: Argument of type '"z"'
                 // parameter of type 'number'.
```

Now if we call `squareOf` with anything but a number, TypeScript will know to complain right away. This is a trivial example (we'll talk a lot more about functions in the next chapter), but it's enough to introduce a couple of concepts that are key to talking about types in TypeScript. We can say the following things about the last code example:

1. `squareOf` 's parameter `n` is *constrained to* `number` .
2. The type of the value `2` is *assignable to* (equivalently: *compatible with*) `number` .

Without a type annotation, `squareOf` is unconstrained in its parameter, and you can pass any type of argument to it. Once we constrain it, TypeScript goes to work for us verifying that every place we call our function, we call it with a compatible argument. In this example the type of `2` is `number` , which is assignable to `squareOf` 's annotation `number` , so TypeScript accepts our code; but `'z'` is a `string` , which is not assignable to `number` , so TypeScript complains.

You can also think of it in terms of *bounds*: we told TypeScript that `n` 's *upper bound* is `number` , so any value we pass to `squareOf` has to be at most a `number` . If it's anything more than a `number` (like, if it's a value that might be a `number` or might be a `string`), then it's not assignable to `n` .

I'll define assignability, bounds, and constraints more formally in Chapter 6. For now, all you need to know is this is the language that we use to talk about whether or not a type can be used in a place where we require a certain type.

The ABCs of Types

Let's take a tour of the types TypeScript supports, what values they contain, and what you can do with them. We'll also cover a few basic language features for working with types: type aliases, union types, and intersection types.

any

`any` is the Godfather of types. It does anything for a price, but you don't want to ask `any` for a favor unless you're completely out of options. In TypeScript everything needs to have a type at compile time, and `any` is the default type when you (the programmer) and TypeScript (the typechecker) can't figure out what type something is. It's a last resort type, and you should avoid it when possible.

Why should you avoid it? Remember what a type is? (It's a set of values and the things you can do with them.) `any` is the set of *all* values, and you can do *anything* with `any`. That means that if you have a value of type `any` you can add to it, multiply by it, call `.pizza()` on it—anything.

`any` makes your value behave like it would in regular JavaScript, and totally prevents the typechecker from working its magic. When you allow `any` into your code you're flying blind. Avoid `any` like fire, and use it only as a very, very last resort.

On the rare occasion that you do need to use it, you do it like this:

```
let a: any = 666             // any
let b: any = ['danger']      // any
let c = a + b                // any
```

Notice how the third type should report an error (why are you trying to add a number and an array?), but doesn't because you told TypeScript that you're adding two `any`s. If you want to use `any`, you have to be explicit about it. When TypeScript infers that some value is of type `any` (for example, if you forgot to annotate a function's parameter, or if you imported an untyped JavaScript module), it will throw a compile-time exception and toss a red squiggly at you in your editor. By explicitly annotating `a` and `b` with the `any` type (`: any`), you avoid the exception—it's your way of telling TypeScript that you know what you're doing.

TSC FLAG: NOIMPLICITANY

By default, TypeScript is permissive, and won't complain about values that it infers as `any`. To get TypeScript to complain about implicit `any`s, be sure to enable the `noImplicitAny` flag in your *tsconfig.json*.

`noImplicitAny` is part of the `strict` family of TSC flags, so if you already enabled `strict` in your *tsconfig.json* (as we did in "tsconfig.json"), you're good to go.

unknown

If `any` is the Godfather, then `unknown` is Keanu Reeves as undercover FBI agent Johnny Utah in *Point Break*: laid back, fits right in with the bad guys, but deep down has a respect for the law and is on the side of the good guys. For the few cases where you have a value whose type you really don't know ahead of time, don't use `any`, and instead reach for `unknown`. Like `any`, it represents any value, but TypeScript won't let you use an `unknown` type until you refine it by checking what it is (see "Refinement").

What operations does `unknown` support? You can compare `unknown` values (with `==`, `===`, `||`, `&&`, and `?`), negate them (with `!`), and refine them (like you can any other type) with JavaScript's `typeof` and `instanceof` operators. Use `unknown` like this:

```
let a: unknown = 30            // unknown
let b = a === 123              // boolean
let c = a + 10                 // Error TS2571: Object is
if (typeof a === 'number') {
  let d = a + 10               // number
}
```

This example should give you a rough idea of how to use `unknown`:

1. TypeScript will never infer something as `unknown`—you have to explicitly annotate it (`a`).[1]
2. You can compare values to values that are of type `unknown` (`b`).
3. But, you can't do things that assume an `unknown` value is of a specific type (`c`); you have to prove to TypeScript that the value really is of that type first (`d`).

boolean

The `boolean` type has two values: `true` and `false`. You can compare them (with `==`, `===`, `||`, `&&`, and `?`), negate them (with `!`), and not much else. Use `boolean` like this:

```
let a = true                   // boolean
var b = false                  // boolean
```

```
const c = true              // true
let d: boolean = true       // boolean
let e: true = true          // true
let f: true = false         // Error TS2322: Type 'fals
                            // to type 'true'.
```

This example shows a few ways to tell TypeScript that something is a `boolean`:

1. You can let TypeScript infer that your value is a `boolean` (`a` and `b`).
2. You can let TypeScript infer that your value is a specific `boolean` (`c`).
3. You can tell TypeScript explicitly that your value is a `boolean` (`d`).
4. You can tell TypeScript explicitly that your value is a specific `boolean` (`e` and `f`).

In general, you will use the first or second way in your programs. Very rarely, you'll use the fourth way—only when it buys you extra type safety (I'll show you examples of that throughout this book). You will almost never use the third way.

The second and fourth cases are particularly interesting because while they do something intuitive, they're supported by surprisingly few programming languages and so might be new to you. What I did in that example was say, "Hey TypeScript! See this variable `e` here? `e` isn't just any old `boolean` —it's the specific `boolean` `true`." By using a value as a type, I essentially limited the possible values for `e` and `f` from all `booleans` to one specific `boolean` each. This feature is called *type literals*.

TYPE LITERAL

A type that represents a single value and nothing else.

In the fourth case I explicitly annotated my variables with type literals, and in the second case TypeScript inferred a literal type for me because I used `const` instead of `let` or `var`. Because TypeScript knows that once a primitive is assigned with `const` its value will never change, it infers the most narrow type it can for that variable. That's why in the second case TypeScript inferred `c`'s type as `true` instead of as `boolean`. To learn more about why TypeScript infers different types for `let` and `const`, jump ahead to “Type Widening”.

We will revisit type literals throughout this book. They are a powerful language feature that lets you squeeze out extra safety all over the place. Type literals make TypeScript unique in the language world and are something you should lord over your Java friends.

number

`number` is the set of all numbers: integers, floats, positives, negatives, `Infinity`, `NaN`, and so on. Numbers can do, well, numbery things, like addition (`+`), subtraction (`-`), modulo (`%`), and comparison (`<`). Let’s look at a few examples:

```
let a = 1234                // number
var b = Infinity * 0.10     // number
const c = 5678              // 5678
let d = a < b               // boolean
let e: number = 100         // number
let f: 26.218 = 26.218      // 26.218
let g: 26.218 = 10          // Error TS2322: Type '10'
                            // to type '26.218'.
```

Like in the `boolean` example, there are four ways to type something as a `number`:

1. You can let TypeScript infer that your value is a `number` (`a` and `b`).
2. You can use `const` so TypeScript infers that your value is a specific `number` (`c`).
3. You can tell TypeScript explicitly that your value is a `number` (`e`).
4. You can tell TypeScript explicitly that your value is a specific `number` (`f` and `g`).

And just like with `booleans`, you’re usually going to let TypeScript infer the type for you (the first way). Once in a while you’ll do some clever programming that requires your number’s type to be restricted to a specific value (the second or fourth way). There is no good reason to explicitly type something as a `number` (the third way).

TIP

When working with long numbers, use numeric separators to make those numbers easier to read. You can use numeric separators in both type and value positions:

```
let oneMillion = 1_000_000 // Equivalent to 1000000
let twoMillion: 2_000_000 = 2_000_000
```

bigint

`bigint` is a newcomer to JavaScript and TypeScript: it lets you work with large integers without running into rounding errors. While the `number` type can only represent whole numbers up to 2^{53}, `bigint` can represent integers bigger than that too. The `bigint` type is the set of all BigInts, and supports things like addition (`+`), subtraction (`-`), multiplication (*), division (`/`), and comparison (`<`). Use it like this:

```
let a = 1234n                 // bigint
const b = 5678n               // 5678n
var c = a + b                 // bigint
let d = a < 1235              // boolean
let e = 88.5n                 // Error TS1353: A bigint l
let f: bigint = 100n          // bigint
let g: 100n = 100n            // 100n
let h: bigint = 100           // Error TS2322: Type '100'
                              // to type 'bigint'.
```

Like with `boolean` and `number`, there are four ways to declare bigints. Try to let TypeScript infer your bigint's type when you can.

WARNING

At the time of writing, `bigint` is not yet natively supported by every JavaScript engine. If your application relies on `bigint`, be careful to check whether or not it's supported by your target platform.

string

`string` is the set of all strings and the things you can do with them like concatenate (`+`), slice (`.slice`), and so on. Let's see some examples:

```
let a = 'hello'             // string
var b = 'billy'             // string
const c = '!'               // '!'
let d = a + ' ' + b + c     // string
let e: string = 'zoom'      // string
let f: 'john' = 'john'      // 'john'
let g: 'john' = 'zoe'       // Error TS2322: Type "zoe'
                            // to type "john".
```

Like `boolean` and `number` , there are four ways to declare `string` types, and you should let TypeScript infer the type for you whenever you can.

symbol

`symbol` is a relatively new language feature that arrived with one of the latest major JavaScript revisions (ES2015). Symbols don't come up often in practice; they are used as an alternative to string keys in objects and maps, in places where you want to be extra sure that people are using the right well-known key and didn't accidentally set the key—think setting a default iterator for your object (`Symbol.iterator`), or overriding at runtime whether or not your object is an instance of something (`Symbol.hasInstance`). Symbols have the type `symbol` , and there isn't all that much you can do with them:

```
let a = Symbol('a')           // symbol
let b: symbol = Symbol('b') // symbol
var c = a === b               // boolean
let d = a + 'x'               // Error TS2469: The '+' op
                              // to type 'symbol'.
```

The way `Symbol('a')` works in JavaScript is by creating a new `symbol` with the given name; that `symbol` is unique, and will not be equal (when compared with `==` or `===`) to any other `symbol` (even if you create a second `symbol` with the same exact name!). Similarly to how the value `27` is inferred to be a `number` when declared with `let` but the specific number `27` when you declare it with `const`, symbols are inferred to be of type `symbol` but can be explicitly typed as `unique symbol`:

```
const e = Symbol('e')                  // typeof e
const f: unique symbol = Symbol('f') // typeof f
let g: unique symbol = Symbol('f')   // Error TS1332: A
                                       // 'unique symbol'
let h = e === e                // boolean
let i = e === f                // Error TS2367: This condi
                               // 'false' since the types
                               // 'unique symbol' have no
```

This example shows off a few ways to create unique symbols:

1. When you declare a new `symbol` and assign it to a `const` variable (not a `let` or `var` variable), TypeScript will infer its type as `unique symbol`. It will show up as `typeof` *yourVariableName*, not `unique symbol`, in your code editor.
2. You can explicitly annotate a `const` variable's type as `unique symbol`.
3. A `unique symbol` is always equal to itself.
4. TypeScript knows at compile time that a `unique symbol` will never be equal to any other `unique symbol`.

Think of `unique symbols` like other literal types, like `1`, `true`, or `"literal"`. They're a way to create a type that represents a particular inhabitant of `symbol`.

Objects

TypeScript's object types specify the shapes of objects. Notably, they can't tell the difference between simple objects (like the kind you make with `{}`) and more complicated ones (the kind you create with `new Blah`). This is by design: JavaScript is generally *structurally typed*, so TypeScript favors that style of programming over a *nominally typed* style.

STRUCTURAL TYPING

A style of programming where you just care that an object has certain properties, and not what its name is (nominal typing). Also called *duck typing* in some languages (or, not judging a book by its cover).

There are a few ways to use types to describe objects in TypeScript. The first is to declare a value as an `object`:

```
let a: object = {
  b: 'x'
}
```

What happens when you access `b`?

```
a.b   // Error TS2339: Property 'b' does not exist on t
```

Wait, that's not very useful! What's the point of typing something as an `object` if you can't do anything with it?

Why, that's a great point, aspiring TypeScripter! In fact, `object` is a little narrower than `any`, but not by much. `object` doesn't tell you a lot about the value it describes, just that the value is a JavaScript object (and that it's not `null`).

What if we leave off the explicit annotation, and let TypeScript do its thing?

```
let a = {
  b: 'x'
}            // {b: string}
a.b          // string

let b = {
  c: {
    d: 'f'
  }
}            // {c: {d: string}}
```

Voilà! You've just discovered the second way to type an object: object literal syntax (not to be confused with type literals). You can either let TypeScript infer your object's shape for you, or explicitly describe it inside curly braces(`{}`):

```
let a: {b: number} = {
  b: 12
}             // {b: number}
```

TYPE INFERENCE WHEN DECLARING OBJECTS WITH CONST

What would have happened if we'd used `const` to declare the object instead?

```
const a: {b: number} = {
  b: 12
}             // Still {b: number}
```

You might be surprised that TypeScript inferred `b` as a `number`, and not as the literal `12`. After all, we learned that when declaring `number`s or `string`s, our choice of `const` or `let` affects how TypeScript infers our types.

Unlike the primitive types we've looked at so far—`boolean`, `number`, `bigint`, `string`, and `symbol`—declaring an object with `const` won't hint to TypeScript to infer its type more narrowly. That's because JavaScript objects are mutable, and for all TypeScript knows you might update their fields after you create them.

We explore this idea more deeply—including how to opt into narrower inference—in "Type Widening".

Object literal syntax says, "Here is a thing that has this shape." The thing might be an object literal, or it might be a class:

```
let c: {
  firstName: string
  lastName: string
} = {
  firstName: 'john',
  lastName: 'barrowman'
}

class Person {
  constructor(
    public firstName: string,   // public is shorthand
                                // this.firstName = fir
    public lastName: string
  ) {}
}
c = new Person('matt', 'smith') // OK
```

`{firstName: string, lastName: string}` describes the *shape* of an object, and both the object literal and the class instance from the last example satisfy that shape, so TypeScript lets us assign a `Person` to `c`.

Let's explore what happens when we add extra properties, or leave out required ones:

```
let a: {b: number}

a = {}  // Error TS2741: Property 'b' is missing in typ
        // but required in type '{b: number}'.

a = {
  b: 1,
  c: 2  // Error TS2322: Type '{b: number; c: number}'
}       // to type '{b: number}'. Object literal may or
        // properties, and 'c' does not exist in type '
```

DEFINITE ASSIGNMENT

This is the first example we've looked at where we first declare a variable (`a`), then initialize it with values (`{}` and `{b: 1, c: 2}`). This is a common JavaScript pattern, and it's supported by TypeScript too.

When you declare a variable in one place and initialize it later, TypeScript will make sure that your variable is *definitely assigned* a value by the time you use it:

```
let i: number
let j = i * 3  // Error TS2454: Variable 'i' is used
               // before being assigned.
```

And don't worry, TypeScript enforces this for you even if you leave off the explicit type annotation:

```
let i
let j = i * 3  // Error TS2532: Object is possibly
               // 'undefined'.
```

By default, TypeScript is pretty strict about object properties—if you say the object should have a property called `b` that's a `number` , TypeScript expects `b` and only `b` . If `b` is missing, or if there are extra properties, TypeScript will complain.

Can you tell TypeScript that something is optional, or that there might be more properties than you planned for? You bet:

```
let a: {
  b: number
                            ❶

  c?: string
                            ❷

  [key: number]: boolean
                            ❸

}
```

`a` has a property `b` that's a `number`.

`a` might have a property `c` that's a `string`. And if `c` is set, it might be `undefined`.

`a` might have any number of numeric properties that are `booleans`.

Let's see what types of objects we can assign to `a`:

```
a = {b: 1}
a = {b: 1, c: undefined}
a = {b: 1, c: 'd'}
a = {b: 1, 10: true}
a = {b: 1, 10: true, 20: false}
a = {10: true}          // Error TS2741: Property 'b' i
                        // '{10: true}'.
a = {b: 1, 33: 'red'}   // Error TS2741: Type 'string'
                        // to type 'boolean'.
```

INDEX SIGNATURES

The `[key: T]: U` syntax is called an *index signature*, and this is the way you tell TypeScript that the given object might contain more keys. The way to read it is, "For this object, all keys of type `T` must have values of type `U`." Index signatures let you safely add more keys to an object, in addition to any keys that you explicitly declared.

There is one rule to keep in mind for index signatures: the index signature key's type (`T`) must be assignable to either `number` or `string`.[2]

Also note that you can use any word for the index signature key's name—it doesn't have to be `key`:

```
let airplaneSeatingAssignments: {
  [seatNumber: string]: string
} = {
  '34D': 'Boris Cherny',
  '34E': 'Bill Gates'
}
```

Optional (`?`) isn't the only modifier you can use when declaring object types. You can also mark fields as read-only (that is, you can declare that a field can't be modified after it's assigned an initial value—kind of like `const` for object properties) with the `readonly` modifier:

```
let user: {
  readonly firstName: string
} = {
  firstName: 'abby'
}

user.firstName // string
user.firstName =
  'abbey with an e' // Error TS2540: Cannot assign to '
                    // is a read-only property.
```

Object literal notation has one special case: empty object types (`{}`). Every type—except `null` and `undefined` —is assignable to an empty object type, which can make it tricky to use. Try to avoid empty object types when possible:

```
let danger: {}
danger = {}
danger = {x: 1}
danger = []
danger = 2
```

As a final note on objects, it's worth mentioning one last way of typing something as an object: `Object` . This is pretty much the same as using `{}` , and is best avoided. [3]

To summarize, there are four ways to declare objects in TypeScript:

1. Object literal notation (like `{a: string}`), also called a *shape*. Use this when you know which fields your object could have, or when all of your object's values will have the same type.
2. Empty object literal notation (`{}`). Try to avoid this.
3. The `object` type. Use this when you just want an object, and don't care about which fields it has.
4. The `Object` type. Try to avoid this.

In your TypeScript programs, you should almost always stick to the first way and the third way. Be careful to avoid the second and fourth ways—use a linter to warn about them, complain about them in code reviews, print posters—use your team's preferred tool to keep them far away from your codebase.

Table 3-1 is a handy reference for options 2–4 in the previous list.

Table 3-1. Is the value a valid object?

Value	`{}`	`object`	`Object`
`{}`	Yes	Yes	Yes
`['a']`	Yes	Yes	Yes
`function () {}`	Yes	Yes	Yes
`new String('a')`	Yes	Yes	Yes
`'a'`	Yes	**No**	Yes
`1`	Yes	**No**	Yes
`Symbol('a')`	Yes	**No**	Yes
`null`	**No**	**No**	**No**
`undefined`	**No**	**No**	**No**

Intermission: Type Aliases, Unions, and Intersections

You are quickly becoming a grizzled TypeScript programmer. You have seen several types and how they work, and are now familiar with the concepts of type systems, types, and safety. It's time we go deeper.

As you know, if you have a value, you can perform certain operations on it, depending on what its type permits. For example, you can use `+` to add two numbers, or `.toUpperCase` to uppercase a string.

If you have a *type*, you can perform some operations on it too. I'm going to introduce a few type-level operations here—there are more to come later in the book, but these are so common that I want to introduce them as early as possible.

Type aliases

Just like you can use variable declarations (`let` , `const` , and `var`) to declare a variable that aliases a value, you can declare a type alias that points to a type. It looks like this:

```
type Age = number

type Person = {
  name: string
  age: Age
}
```

`Age` is but a `number` . It can also help make the definition of the `Person` shape easier to understand. Aliases are never inferred by TypeScript, so you have to type them explicitly:

```
let age: Age = 55

let driver: Person = {
  name: 'James May'
  age: age
}
```

Because `Age` is just an alias for `number` , that means it's also assignable to `number` , so we can rewrite this as:

```
let age = 55

let driver: Person = {
  name: 'James May'
  age: age
}
```

Wherever you see a type alias used, you can substitute in the type it aliases without changing the meaning of your program.

Like JavaScript variable declarations (`let` , `const` , and `var`), you can't declare a type twice:

```
type Color = 'red'
type Color = 'blue'  // Error TS2300: Duplicate identif
```

And like `let` and `const` , type aliases are block-scoped. Every block and every function has its own scope, and inner type alias declarations shadow outer ones:

```
type Color = 'red'

let x = Math.random() < .5

if (x) {
  type Color = 'blue'  // This shadows the Color declar
  let b: Color = 'blue'
} else {
  let c: Color = 'red'
}
```

Type aliases are useful for DRYing up repeated complex types,[4] and for making it clear what a variable is used for (some people prefer descriptive type names to descriptive variable names!). When deciding whether or not to alias a type, use the same judgment as when deciding whether or not to pull a value out into its own variable.

Union and intersection types

If you have two things `A` and `B` , the *union* of those things is their sum (everything in `A` or `B` or both), and the *intersection* is what they have in common (everything in both `A` and `B`). The easiest way to think about this is with sets. In Figure 3-2 I represent sets as circles. On the left is the union, or *sum*, of the two sets; on the right is their intersection, or *product*.

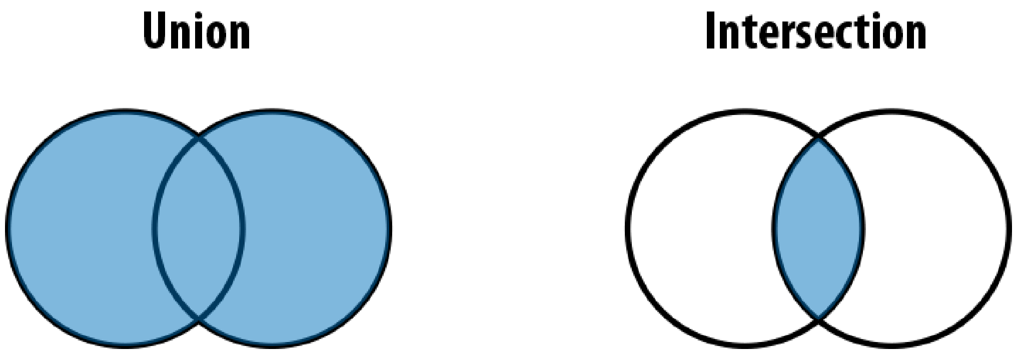

Figure 3-2. Union (|) and intersection (&)

TypeScript gives us special type operators to describe unions and intersections of types: `|` for union and `&` for intersection. Since types are a lot like sets, we can think of them in the same way:

```
type Cat = {name: string, purrs: boolean}
type Dog = {name: string, barks: boolean, wags: boolear
type CatOrDogOrBoth = Cat | Dog
type CatAndDog = Cat & Dog
```

If something is a `CatOrDogOrBoth`, what do you know about it? You know that it has a `name` property that's a string, and not much else. On the flip side, what can you assign to a `CatOrDogOrBoth`? Well, a `Cat`, a `Dog`, or both:

```
// Cat
let a: CatOrDogOrBoth = {
  name: 'Bonkers',
  purrs: true
}

// Dog
a = {
  name: 'Domino',
  barks: true,
  wags: true
}

// Both
a = {
  name: 'Donkers',
  barks: true,
  purrs: true,
  wags: true
}
```

This is worth reiterating: a value with a union type (|) isn't necessarily one specific member of your union; in fact, it can be both members at once![5]

On the other hand, what do you know about `CatAndDog`? Not only does your canine-feline hybrid super-pet have a `name`, but it can purr, bark, and wag:

```
let b: CatAndDog = {
  name: 'Domino',
  barks: true,
  purrs: true,
  wags: true
}
```

Unions come up naturally a lot more often than intersections do. Take this function, for example:

```
function trueOrNull(isTrue: boolean) {
  if (isTrue) {
    return 'true'
  }
  return null
}
```

What is the type of the value this function returns? Well, it might be a `string`, or it might be `null`. We can express its return type as:

```
type Returns = string | null
```

How about this one?

```
function(a: string, b: number) {
  return a || b
}
```

If `a` is truthy then the return type is `string`, and otherwise it's `number`: in other words, `string | number`.

The last place where unions come up naturally is in arrays (specifically the heterogeneous kind), which we'll talk about next.

Arrays

Like in JavaScript, TypeScript arrays are special kinds of objects that support things like concatenation, pushing, searching, and slicing. It's example time:

```
let a = [1, 2, 3]            // number[]
var b = ['a', 'b']           // string[]
let c: string[] = ['a']      // string[]
let d = [1, 'a']             // (string | number)[]
const e = [2, 'b']           // (string | number)[]

let f = ['red']
f.push('blue')
f.push(true)                 // Error TS2345: Argument c
                             // assignable to parameter

let g = []                   // any[]
g.push(1)                    // number[]
g.push('red')                // (string | number)[]

let h: number[] = []         // number[]
h.push(1)                    // number[]
h.push('red')                // Error TS2345: Argument c
                             // assignable to parameter
```

NOTE

TypeScript supports two syntaxes for arrays: `T[]` and `Array<T>`. They are identical both in meaning and in performance. This book uses `T[]` syntax for its terseness, but you should pick whichever style you like for your own code.

As you read through these examples, notice that everything but `c` and `h` is implicitly typed. You'll also notice that TypeScript has rules about what you can and can't put in an array.

The general rule of thumb is to keep arrays *homogeneous*. That is, don't mix apples and oranges and numbers in a single array—try to design your programs so that every element of your array has the same type. The reason is that otherwise, you're going to have to do more work to prove to TypeScript that what you're doing is safe.

To see why things are easier when your arrays are homogeneous, take a look at example `f`. I initialized an array with the string `'red'` (at the point when I declared the array it contained just strings, so TypeScript inferred that it must be an array of strings). I then pushed `'blue'` onto it; `'blue'` is a string, so TypeScript let it pass. Then I tried to push `true` onto the array, but that failed! Why? Because `f` is an array of strings, and `true` is not a string.

On the other hand, when I initialized `d` I gave it a `number` and a `string`, so TypeScript inferred that it must be an array of type `number | string`. Because each element might be either a number or a string, you have to check which it is before using it. For example, say you want to map over that array, converting every letter to uppercase and tripling every number:

```
let d = [1, 'a']

d.map(_ => {
  if (typeof _ === 'number') {
    return _ * 3
  }
  return _.toUpperCase()
})
```

You have to query the type of each item with `typeof`, checking if it's a `number` or a `string` before you can do anything with it.

Like with objects, creating arrays with `const` won't hint to TypeScript to infer their types more narrowly. That's why TypeScript inferred both `d` and `e` to be arrays of `number | string`.

`g` is the special case: when you initialize an empty array, TypeScript doesn't know what type the array's elements should be, so it gives you the benefit of the doubt and makes them `any`. As you manipulate the array and add elements to it, TypeScript starts to piece together your array's type. Once your array leaves the scope it was defined in (for example, if you declared it in a function, then returned it), TypeScript will assign it a final type that can't be expanded anymore:

```
function buildArray() {
  let a = []                 // any[]
  a.push(1)                  // number[]
  a.push('x')                // (string | number)[]
```

```
    return a
  }

  let myArray = buildArray()  // (string | number)[]
  myArray.push(true)          // Error 2345: Argument of
                              // assignable to parameter
```

So as far as uses of `any` go, this one shouldn't make you sweat too much.

Tuples

Tuples are subtypes of `array`. They're a special way to type arrays that have fixed lengths, where the values at each index have specific, known types. Unlike most other types, tuples have to be explicitly typed when you declare them. That's because the JavaScript syntax is the same for tuples and arrays (both use square brackets), and TypeScript already has rules for inferring array types from square brackets:

```
let a: [number] = [1]

// A tuple of [first name, last name, birth year]
let b: [string, string, number] = ['malcolm', 'gladwell

b = ['queen', 'elizabeth', 'ii', 1926]  // Error TS2322
                                        // assignable t
```

Tuples support optional elements too. Just like in object types, `?` means "optional":

```
// An array of train fares, which sometimes vary depenc
let trainFares: [number, number?][] = [
  [3.75],
  [8.25, 7.70],
  [10.50]
]

// Equivalently:
let moreTrainFares: ([number] | [number, number])[] = [
  // ...
]
```

Tuples also support rest elements, which you can use to type tuples with minimum lengths:

```
// A list of strings with at least 1 element
let friends: [string, ...string[]] = ['Sara', 'Tali', '

// A heterogeneous list
let list: [number, boolean, ...string[]] = [1, false, '
```

Not only do tuple types safely encode heterogeneous lists, but they also capture the length of the list they type. These features buy you significantly more safety than plain old arrays—use them often.

Read-only arrays and tuples

While regular arrays are mutable (meaning you can `.push` onto them, `.splice` them, and update them in place), which is probably what you want most of the time, sometimes you want an immutable array—one that you can update to produce a new array, leaving the original unchanged.

TypeScript comes with a `readonly` array type out of the box, which you can use to create immutable arrays. Read-only arrays are just like regular arrays, but you can’t update them in place. To create a read-only array, use an explicit type annotation; to update a read-only array, use nonmutating methods like `.concat` and `.slice` instead of mutating ones like `.push` and `.splice`:

```
let as: readonly number[] = [1, 2, 3]     // readonly r
let bs: readonly number[] = as.concat(4)  // readonly r
let three = bs[2]                         // number
as[4] = 5            // Error TS2542: Index signature i
                     // 'readonly number[]' only permit
as.push(6)           // Error TS2339: Property 'push' d
                     // exist on type 'readonly number[
```

Like `Array`, TypeScript comes with a couple of longer-form ways to declare read-only arrays and tuples:

```
type A = readonly string[]          // readonly string
type B = ReadonlyArray<string>      // readonly string
type C = Readonly<string[]>         // readonly string

type D = readonly [number, string]  // readonly [numbe
type E = Readonly<[number, string]> // readonly [numbe
```

Which syntax you use—the terser `readonly` modifier, or the longer-form `Readonly` or `ReadonlyArray` utilities—is a matter of taste.

Note that while read-only arrays can make your code easier to reason about in some cases by avoiding mutability, they are backed by regular JavaScript arrays. That means even small updates to an array result in having to copy the original array first, which can hurt your application's runtime performance if you're not careful. For small arrays this overhead is rarely noticeable, but for bigger arrays, the overhead can become significant.

TIP

If you plan to make heavy use of immutable arrays, consider reaching for a more efficient implementation, like Lee Byron's excellent `immutable`.

---

null, undefined, void, and never

JavaScript has two values to represent an absence of something: `null` and `undefined`. TypeScript supports both of these as values, and it also has types for them—any guess what they're called? You got it, the types are called `null` and `undefined` too.

They're both special types, because in TypeScript the only thing of type `undefined` is the value `undefined`, and the only thing of type `null` is the value `null`.

JavaScript programmers usually use the two interchangeably, though there is a subtle semantic difference worth mentioning: `undefined` means that something hasn't been defined yet, and `null` means an absence of a value (like if you tried to compute a value, but ran into an error along the way). These are just conventions and TypeScript doesn't hold you to them, but it can be a useful distinction to make.

In addition to `null` and `undefined`, TypeScript also has `void` and `never`. These are really specific, special-purpose types that draw even finer lines between the different kinds of things that don't exist: `void` is the return type of a function that doesn't explicitly return anything (for example, `console.log`), and `never` is the type of a function that never returns at all (like a function that throws an exception, or one that runs forever):

```
// (a) A function that returns a number or null
function a(x: number) {
  if (x < 10) {
    return x
  }
  return null
}

// (b) A function that returns undefined
function b() {
  return undefined
}

// (c) A function that returns void
function c() {
  let a = 2 + 2
  let b = a * a
}

// (d) A function that returns never
function d() {
  throw TypeError('I always error')
}

// (e) Another function that returns never
function e() {
  while (true) {
    doSomething()
  }
}
```

(a) and (b) explicitly return `null` and `undefined`, respectively. (c) returns `undefined`, but it doesn't do so with an explicit `return` statement, so we say it returns `void`. (d) throws an exception, and (e) runs forever—neither will ever return, so we say their return type is `never`.

If `unknown` is the supertype of every other type, then `never` is the subtype of every other type. We call it a *bottom type*. That means it's assignable to every other type, and a value of type `never` can be used anywhere safely. This has mostly theoretical significance,[6] but is something that will come up when you talk about TypeScript with other language nerds.

Table 3-2 summarizes how the four absence types are used.

Table 3-2. Types that mean an absence of something

Type	**Meaning**
`null`	Absence of a value
`undefined`	Variable that has not been assigned a value yet
`void`	Function that doesn't have a `return` statement
`never`	Function that never returns

STRICT NULL CHECKING

In older versions of TypeScript (or with TSC's `strictNullChecks` option set to `false`), `null` behaves a little differently: it is a subtype of all types, except `never`. That means every type is nullable, and you can never really trust the type of anything without first checking if it's `null` or not. For example, if someone passes the variable `pizza` to your function and you want to call the method `.addAnchovies` on it, you first have to check if your `pizza` is `null` before you can add delicious tiny fish to it. In practice this is really tedious to do with every single variable, so people often forget to actually check first. Then, when something really is `null`, you get a dreaded null pointer exception at runtime:

```
function addDeliciousFish(pizza: Pizza) {
  return pizza.addAnchovies()  // Uncaught TypeError: C
}                              // property 'addAnchovie

// TypeScript lets this fly with strictNullChecks = fal
addDeliciousFish(null)
```

`null` has been called the "billion dollar mistake" by the guy that introduced it in the 1960s. The problem with `null` is it's something that most languages' type systems can't express and don't check for; so when a programmer tries to do something with a variable that they thought was defined but it actually turns out to be `null` at runtime, the code throws a runtime exception!

Why? Don't ask me, I'm just the guy writing this book. But languages are coming around to encoding `null` in their type systems, and TypeScript is a great example of how to do it right. If the goal is to catch as many bugs as possible at compile time before your users encounter them, then being able to check for `null` in the type system is indispensable.

Enums

Enums are a way to *enumerate* the possible values for a type. They are unordered data structures that map keys to values. Think of them like objects where the keys are fixed at compile time, so TypeScript can check that the given key actually exists when you access it.

There are two kinds of enums: enums that map from strings to strings, and enums that map from strings to numbers. They look like this:

```
enum Language {
  English,
  Spanish,
  Russian
}
```

NOTE

By convention, enum names are uppercase and singular. Their keys are also uppercase.

TypeScript will automatically infer a number as the value for each member of your enum, but you can also set values explicitly. Let's make explicit what TypeScript inferred in the previous example:

```
enum Language {
  English = 0,
  Spanish = 1,
  Russian = 2
}
```

To retrieve a value from an enum, you access it with either dot or bracket notation—just like you would to get a value from a regular object:

```
let myFirstLanguage = Language.Russian      // Language
let mySecondLanguage = Language['English']  // Language
```

You can split your enum across multiple declarations, and TypeScript will automatically merge them for you (to learn more, jump ahead to "Declaration Merging"). Beware that when you do split your enum, TypeScript can only infer values for one of those declarations, so it's good practice to explicitly assign a value to each enum member:

```
enum Language {
  English = 0,
  Spanish = 1
}

enum Language {
  Russian = 2
}
```

You can use computed values, and you don't have to define all of them (TypeScript will do its best to infer what's missing):

```
enum Language {
  English = 100,
  Spanish = 200 + 300,
  Russian                  // TypeScript infers 501 (the
}
```

You can also use string values for enums, or even mix string and number values:

```
enum Color {
  Red = '#c10000',
  Blue = '#007ac1',
  Pink = 0xc10050,         // A hexadecimal literal
  White = 255              // A decimal literal
}

let red = Color.Red        // Color
let pink = Color.Pink      // Color
```

TypeScript lets you access enums both by value and by key for convenience, but this can get unsafe quickly:

```
let a = Color.Red          // Color
let b = Color.Green        // Error TS2339: Property 'Gr
                           // on type 'typeof Color'.
let c = Color[0]           // string
let d = Color[6]           // string (!!!)
```

You shouldn't be able to get `Color[6]`, but TypeScript doesn't stop you! We can ask TypeScript to prevent this kind of unsafe access by opting into a safer subset of enum behavior with `const enum` instead. Let's rewrite our `Language` enum from earlier:

```
const enum Language {
  English,
  Spanish,
  Russian
}

// Accessing a valid enum key
let a = Language.English  // Language

// Accessing an invalid enum key
let b = Language.Tagalog  // Error TS2339: Property 'Ta
                          // on type 'typeof Language'.

// Accessing a valid enum value
let c = Language[0]       // Error TS2476: A const enum
                          // accessed using a string li

// Accessing an invalid enum value
let d = Language[6]       // Error TS2476: A const enum
                          // accessed using a string li
```

A `const enum` doesn't let you do reverse lookups, and so behaves a lot like a regular JavaScript object. It also doesn't generate any JavaScript code by default, and instead inlines the enum member's value wherever it's used (for example, TypeScript will replace every occurrence of `Language.Spanish` with its value, `1`).

TSC FLAG: PRESERVECONSTENUMS

`const enum` inlining can lead to safety issues when you import a `const enum` from someone else's TypeScript code: if the enum author updates their `const enum` after you've compiled your TypeScript code, then your version of the enum and their version might point to different values at runtime, and TypeScript will be none the wiser.

If you use `const enum` s, be careful to avoid inlining them and to only use them in TypeScript programs that you control: avoid using them in programs that you're planning to publish to NPM, or to make available for others to use as a library.

To enable runtime code generation for `const enum` s, switch the `preserveConstEnums` TSC setting to `true` in your *tsconfig.json*:

```
{
  "compilerOptions": {
    "preserveConstEnums": true
  }
}
```

Let's see how we use `const enum` s:

```
const enum Flippable {
  Burger,
  Chair,
  Cup,
  Skateboard,
  Table
}

function flip(f: Flippable) {
  return 'flipped it'
}

flip(Flippable.Chair)     // 'flipped it'
flip(Flippable.Cup)       // 'flipped it'
flip(12)                  // 'flipped it' (!!!)
```

Everything looks great— `Chairs` and `Cups` work exactly as you expect… until you realize that all numbers are also assignable to enums! That behavior is an unfortunate consequence of TypeScript's assignability rules, and to fix it you have to be extra careful to only use string-valued enums:

```
const enum Flippable {
  Burger = 'Burger',
  Chair = 'Chair',
  Cup = 'Cup',
  Skateboard = 'Skateboard',
  Table = 'Table'
}

function flip(f: Flippable) {
  return 'flipped it'
}

flip(Flippable.Chair)     // 'flipped it'
flip(Flippable.Cup)       // 'flipped it'
flip(12)                  // Error TS2345: Argument of
                          // assignable to parameter of
flip('Hat')               // Error TS2345: Argument of
                          // assignable to parameter of
```

All it takes is one pesky numeric value in your enum to make the whole enum unsafe.

WARNING

Because of all the pitfalls that come with using enums safely, I recommend you stay away from them—there are plenty of better ways to express yourself in TypeScript.

And if a coworker insists on using enums and there's nothing you can do to change their mind, be sure to ninja-merge a few TSLint rules while they're out to warn about numeric values and non- `const` enums.

Summary

In short, TypeScript comes with a bunch of built-in types. You can let TypeScript infer types for you from your values, or you can explicitly type your values. `const` will infer more specific types, `let` and `var` more general ones. Most types have general and more specific counterparts, the latter subtypes of the former (see Table 3-3).

Table 3-3. Types and their more specific subtypes

Type	**Subtype**
`boolean`	Boolean literal
`bigint`	BigInt literal
`number`	Number literal
`string`	String literal
`symbol`	`unique symbol`
`object`	Object literal
Array	Tuple
`enum`	`const enum`

Exercises

1. For each of these values, what type will TypeScript infer?
   1. `let a = 1042`
   2. `let b = 'apples and oranges'`
   3. `const c = 'pineapples'`
   4. `let d = [true, true, false]`
   5. `let e = {type: 'ficus'}`
   6. `let f = [1, false]`
   7. `const g = [3]`
   8. `let h = null` (try this out in your code editor, then jump ahead to "Type Widening" if the result surprises you!)
2. Why does each of these throw the error it does?
   1.
      ```
      let i: 3 = 3
      ```

```
i = 4 // Error TS2322: Type '4' is not assignable
```

2.
```
let j = [1, 2, 3]
j.push(4)
j.push('5') // Error TS2345: Argument of type '"5
            // assignable to parameter of type 'n
```

3.
```
let k: never = 4 // Error TSTS2322: Type '4' is n
                 // to type 'never'.
```

4.
```
let l: unknown = 4
let m = l * 2 // Error TS2571: Object is of type
```

1 Almost. When `unknown` is part of a union type, the result of the union will be `unknown`. You'll read more about union types in "Union and intersection types".

2 Objects in JavaScript use strings for keys; arrays are special kinds of objects that use numerical keys.

3 There's one minor technical difference: `{}` lets you define whatever types you want for built-in methods on the `Object` prototype, like `.toString` and `.hasOwnProperty` (head over to MDN to learn more about prototypes), while `Object` enforces that the types you declare are assignable to those on `Object`'s prototype. For example, this code typechecks: `let a: {} = {toString() { return 3 }}`. But if you change the type annotation to `Object`, TypeScript complains: `let b: Object = {toString() { return 3 }}` results in `Error TS2322: Type 'number' is not assignable to type 'string'`.

4 The acronym DRY stands for "Don't Repeat Yourself"—the idea that code shouldn't be repetitive. It was introduced by Andrew Hunt and David Thomas in their book *The Pragmatic Programmer: From Journeyman to Master* (Addison-Wesley).

5 Jump ahead to "Discriminated union types" to learn how to hint to TypeScript that your union is disjoint and a value of that union's type has to be one or the other, and not both.

6 The way to think about a bottom type is as a type that has no values. A bottom type corresponds to a mathematical proposition that’s always false.