

## About this Handbook

Over 20 years after its introduction to the programming community, JavaScript is now one of the most widespread cross-platform languages ever created. Starting as a small scripting language for adding trivial interactivity to webpages, JavaScript has grown to be a language of choice for both frontend and backend applications of every size. While the size, scope, and complexity of programs written in JavaScript has grown exponentially, the ability of the JavaScript language to express the relationships between different units of code has not. Combined with JavaScript's rather peculiar runtime semantics, this mismatch between language and program complexity has made JavaScript development a difficult task to manage at scale.

The most common kinds of errors that programmers write can be described as type errors: a certain kind of value was used where a different kind of value was expected. This could be due to simple typos, a failure to understand the API surface of a library, incorrect assumptions about runtime behavior, or other errors. The goal of TypeScript is to be a static typechecker for JavaScript programs - in other words, a tool that runs before your code runs (static) and ensures that the types of the program are correct (typechecked).

If you are coming to TypeScript without a JavaScript background, with the intention of TypeScript being your first language, we recommend you first start reading the documentation on either the Microsoft Learn JavaScript tutorial or read JavaScript at the Mozilla Web Docs.

If you have experience in other languages, you should be able to pick up JavaScript syntax quite quickly by reading the handbook.

How is this Handbook Structured

The handbook is split into two sections:

## The Handbook

The TypeScript Handbook is intended to be a comprehensive document that explains TypeScript to everyday programmers. You can read the handbook by going from top to bottom in the left-hand navigation.

You should expect each chapter or page to provide you with a strong understanding of the given concepts. The TypeScript Handbook is not a complete language specification, but it is intended to be a comprehensive guide to all of the language's features and behaviors.

A reader who completes the walkthrough should be able to:

Read and understand commonly-used TypeScript syntax and patterns

Explain the effects of important compiler options

Correctly predict type system behavior in most cases

In the interests of clarity and brevity, the main content of the Handbook will not explore every edge case or minutiae of the features being covered. You can find more details on particular concepts in the reference articles.

## Reference Files

The reference section below the handbook in the navigation is built to provide a richer understanding of how a particular part of TypeScript works. You can read it top-to-bottom, but each section aims to provide a deeper explanation of a single concept - meaning there is no aim for continuity.

## Non-Goals

The Handbook is also intended to be a concise document that can be comfortably read in a few hours. Certain topics won't be covered in order to keep things short.

Specifically, the Handbook does not fully introduce core JavaScript basics like functions, classes, and closures. Where appropriate, we'll include links to background reading that you can use to read up on those concepts.

The Handbook also isn't intended to be a replacement for a language specification. In some cases, edge cases or formal descriptions of behavior will be skipped in favor of high-level, easier-to-understand explanations. Instead, there are separate reference pages that more precisely and formally describe many aspects of TypeScript's behavior. The reference pages are not intended for readers unfamiliar with TypeScript, so they may use advanced terminology or reference topics you haven't read about yet.

Finally, the Handbook won't cover how TypeScript interacts with other tools, except where necessary. Topics like how to configure TypeScript with webpack, rollup, parcel, react, babel, closure, lerna, rush, bazel, preact, vue, angular, svelte, jquery, yarn, or npm are out of scope - you can find these resources elsewhere on the web.

## Get Started

Before getting started with The Basics, we recommend reading one of the following introductory pages. These introductions are intended to highlight key similarities and differences between TypeScript and your favored programming language, and clear up common misconceptions specific to those languages.

TypeScript for the New Programmer

TypeScript for JavaScript Programmers

TypeScript for Java/C# Programmers

TypeScript for Functional Programmers

Otherwise, jump to The Basics.

Each and every value in JavaScript has a set of behaviors you can observe from running different operations.

That sounds abstract, but as a quick example, consider some operations we might run on a variable named `message`.

```
js// Accessing the property 'toLowerCase' on 'message' and then calling  
message.toLowerCase();// Calling 'message' message();
```

If we break this down, the first runnable line of code accesses a property called `toLowerCase` and then calls it.

The second one tries to call `message` directly.

But assuming we don't know the value of `message` - and that's pretty common - we can't reliably say what results we'll get from trying to run any of this code.

The behavior of each operation depends entirely on what value we had in the first place.

Is message callable?

Does it have a property called `toLowerCase` on it?

If it does, is `toLowerCase` even callable?

If both of these values are callable, what do they return?

The answers to these questions are usually things we keep in our heads when we write JavaScript, and we have to hope we got all the details right.

Let's say `message` was defined in the following way.

```
jsconst message = "Hello World!";
```

As you can probably guess, if we try to run `message.toLowerCase()`, we'll get the same string only in lower-case.

What about that second line of code?

If you're familiar with JavaScript, you'll know this fails with an exception:

```
txtTypeError: message is not a function
```

It'd be great if we could avoid mistakes like this.

When we run our code, the way that our JavaScript runtime chooses what to do is by figuring out the type of the value - what sorts of behaviors and capabilities it has.

That's part of what that `TypeError` is alluding to - it's saying that the string "Hello World!" cannot be called as a function.

For some values, such as the primitives string and number, we can identify their type at runtime using the `typeof` operator.

But for other things like functions, there's no corresponding runtime mechanism to identify their types.

For example, consider this function:

```
jsfunction fn(x) { return x.flip();}
```

We can observe by reading the code that this function will only work if given an object with a callable `flip` property, but JavaScript doesn't surface this information in a way that we can check while the code is running.

The only way in pure JavaScript to tell what `fn` does with a particular value is to call it and see what happens.

This kind of behavior makes it hard to predict what the code will do before it runs, which means it's harder to know what your code is going to do while you're writing it.

Seen in this way, a type is the concept of describing which values can be passed to `fn` and which will crash.

JavaScript only truly provides dynamic typing - running the code to see what happens.

The alternative is to use a static type system to make predictions about what code is expected before it runs.

Static type-checking

Think back to that `TypeError` we got earlier from trying to call a string as a function.

Most people don't like to get any sorts of errors when running their code - those are considered bugs!

And when we write new code, we try our best to avoid introducing new bugs.

If we add just a bit of code, save our file, re-run the code, and immediately see the error, we might be able to isolate the problem quickly; but that's not always the case.

We might not have tested the feature thoroughly enough, so we might never actually run into a potential error that would be thrown!

Or if we were lucky enough to witness the error, we might have ended up doing large refactorings and adding a lot of different code that we're forced to dig through.

Ideally, we could have a tool that helps us find these bugs before our code runs.

That's what a static type-checker like TypeScript does.

Static types systems describe the shapes and behaviors of what our values will be when we run our programs.

A type-checker like TypeScript uses that information and tells us when things might be going off the rails.

```
tsconst message = "hello!"; message();
```

This expression is not callable.

Type 'String' has no call signatures.<sup>2349</sup>

Type 'String' has no call signatures.Try

Running that last sample with TypeScript will give us an error message before we run the code in the first place.

### Non-exception Failures

So far we've been discussing certain things like runtime errors - cases where the JavaScript runtime tells us that it thinks something is nonsensical.

Those cases come up because the ECMAScript specification has explicit instructions on how the language should behave when it runs into something unexpected.

For example, the specification says that trying to call something that isn't callable should throw an error.

Maybe that sounds like "obvious behavior", but you could imagine that accessing a property that doesn't exist on an object should throw an error too.

Instead, JavaScript gives us different behavior and returns the value undefined:

```
jsconst user = { name: "Daniel", age: 26,}; user.location;
```

// returns undefined

Ultimately, a static type system has to make the call over what code should be flagged as an error in its system, even if it's "valid" JavaScript that won't immediately throw an error.

In TypeScript, the following code produces an error about location not being defined:

```
tsconst user = { name: "Daniel", age: 26,}; user.location;
```

Property 'location' does not exist on type '{ name: string; age: number; }'.<sup>2339</sup>

While sometimes that implies a trade-off in what you can express, the intent is to catch legitimate bugs in our programs.

And TypeScript catches a lot of legitimate bugs.

For example: typos,

```
tsconst announcement = "Hello World!";
```

// How quickly can you spot the typos?

```
announcement.toLocaleLowerCase(); announcement.toLocalLowerCase();
```

// We

probably meant to write this...`announcement.toLocaleLowerCase();`

uncalled functions,

```
tsfunction flipCoin() { // Meant to be Math.random() return Math.random <
```

```
0.5; Operator '<' cannot be applied to types '() => number' and 'number'.2365
```

```
Operator '<' cannot be applied to types '() => number' and 'number'.}
```

Try

or basic logic errors.

```
tsconst value = Math.random() < 0.5 ? "a" : "b"; if (value !== "a") { // ...} else if (value
```

```
=== "b") {This comparison appears to be unintentional because the types "a" and "b"
have no overlap.2367This comparison appears to be unintentional because the types
"a" and "b" have no overlap. // Oops, unreachable}Try
```

## Types for Tooling

TypeScript can catch bugs when we make mistakes in our code.

That's great, but TypeScript can also prevent us from making those mistakes in the first place.

The type-checker has information to check things like whether we're accessing the right properties on variables and other properties.

Once it has that information, it can also start suggesting which properties you might want to use.

That means TypeScript can be leveraged for editing code too, and the core type-checker can provide error messages and code completion as you type in the editor.

That's part of what people often refer to when they talk about tooling in TypeScript.

```
tsimport express from "express";const app = express(); app.get("/", function (req, res)
{ res.send    sendDatesendFilesendStatus}); app.listen(3000);Try
TypeScript takes tooling seriously, and that goes beyond completions and errors as you
type.
```

An editor that supports TypeScript can deliver “quick fixes” to automatically fix errors, refactorings to easily re-organize code, and useful navigation features for jumping to definitions of a variable, or finding all references to a given variable.

All of this is built on top of the type-checker and is fully cross-platform, so it's likely that your favorite editor has TypeScript support available.

tsc, the TypeScript compiler

We've been talking about type-checking, but we haven't yet used our type-checker.

Let's get acquainted with our new friend tsc, the TypeScript compiler.

First we'll need to grab it via npm.

```
shnpm install -g typescript
```

This installs the TypeScript Compiler tsc globally.

You can use npx or similar tools if you'd prefer to run tsc from a local node\_modules package instead.

Now let's move to an empty folder and try writing our first TypeScript program: hello.ts:

```
ts// Greets the world.console.log("Hello world!");Try
```

Notice there are no frills here; this “hello world” program looks identical to what you'd write for a “hello world” program in JavaScript.

And now let's type-check it by running the command tsc which was installed for us by the typescript package.

```
shhtsc hello.ts
```

Tada!

Wait, “tada” what exactly?

We ran tsc and nothing happened!

Well, there were no type errors, so we didn't get any output in our console since there was nothing to report.

But check again - we got some file output instead.

If we look in our current directory, we'll see a hello.js file next to hello.ts.

That's the output from our hello.ts file after tsc compiles or transforms it into a plain JavaScript file.

And if we check the contents, we'll see what TypeScript spits out after it processes a .ts file:

```
js// Greets the world.console.log("Hello world!");
```

In this case, there was very little for TypeScript to transform, so it looks identical to what we wrote.

The compiler tries to emit clean readable code that looks like something a person would write.

While that's not always so easy, TypeScript indents consistently, is mindful of when our code spans across different lines of code, and tries to keep comments around.

What about if we did introduce a type-checking error?

Let's rewrite hello.ts:

```
ts// This is an industrial-grade general-purpose greeter function:function greet(person, date) { console.log(`Hello ${person}, today is ${date}!`);} greet("Brendan");
```

Try If we run tsc hello.ts again, notice that we get an error on the command line!

```
txtExpected 2 arguments, but got 1.
```

TypeScript is telling us we forgot to pass an argument to the greet function, and rightfully so.

So far we've only written standard JavaScript, and yet type-checking was still able to find problems with our code.

Thanks TypeScript!

Emitting with Errors

One thing you might not have noticed from the last example was that our hello.js file changed again.

If we open that file up then we'll see that the contents still basically look the same as our input file.

That might be a bit surprising given the fact that tsc reported an error about our code, but this is based on one of TypeScript's core values: much of the time, you will know better than TypeScript.

To reiterate from earlier, type-checking code limits the sorts of programs you can run, and so there's a tradeoff on what sorts of things a type-checker finds acceptable.

Most of the time that's okay, but there are scenarios where those checks get in the way.

For example, imagine yourself migrating JavaScript code over to TypeScript and introducing type-checking errors.

Eventually you'll get around to cleaning things up for the type-checker, but that original JavaScript code was already working!

Why should converting it over to TypeScript stop you from running it?

So TypeScript doesn't get in your way.

Of course, over time, you may want to be a bit more defensive against mistakes, and make TypeScript act a bit more strictly.

In that case, you can use the noEmitOnError compiler option.

Try changing your hello.ts file and running tsc with that flag:

```
shtsc --noEmitOnError hello.ts
```

You'll notice that hello.js never gets updated.

### Explicit Types

Up until now, we haven't told TypeScript what person or date are.

Let's edit the code to tell TypeScript that person is a string, and that date should be a Date object.

We'll also use the toString() method on date.

```
tsfunction greet(person: string, date: Date) { console.log(` Hello ${person}, today is  
${date.toString()}!`); } Try
```

What we did was add type annotations on person and date to describe what types of values greet can be called with.

You can read that signature as "greet takes a person of type string, and a date of type Date".

With this, TypeScript can tell us about other cases where greet might have been called incorrectly.

For example...

```
tsfunction greet(person: string, date: Date) { console.log(` Hello ${person}, today is  
${date.toString()}!`); } greet("Maddison", Date()); Argument of type 'string' is not  
assignable to parameter of type 'Date'.2345 Argument of type 'string' is not assignable  
to parameter of type 'Date'. Try
```

Huh?

TypeScript reported an error on our second argument, but why?

Perhaps surprisingly, calling Date() in JavaScript returns a string.

On the other hand, constructing a Date with new Date() actually gives us what we were expecting.

Anyway, we can quickly fix up the error:

```
tsfunction greet(person: string, date: Date) { console.log(` Hello ${person}, today is  
${date.toString()}!`); } greet("Maddison", new Date()); Try
```

Keep in mind, we don't always have to write explicit type annotations.

In many cases, TypeScript can even just infer (or "figure out") the types for us even if we omit them.

```
tslet msg = "hello there!"; let msg: string Try
```

Even though we didn't tell TypeScript that msg had the type string it was able to figure that out.

That's a feature, and it's best not to add annotations when the type system would end up inferring the same type anyway.

Note: The message bubble inside the previous code sample is what your editor would show if you had hovered over the word.

### Erased Types

Let's take a look at what happens when we compile the above function greet with tsc to output JavaScript:

```
ts"use strict";function greet(person, date) { console.log("Hello ".concat(person, "  
today is ").concat(date.toString(), "!")); }greet("Maddison", new Date()); Try
```

Notice two things here:

Our person and date parameters no longer have type annotations. Our “template string” - that string that used backticks (the ` character) - was converted to plain strings with concatenations.

More on that second point later, but let’s now focus on that first point. Type annotations aren’t part of JavaScript (or ECMAScript to be pedantic), so there really aren’t any browsers or other runtimes that can just run TypeScript unmodified. That’s why TypeScript needs a compiler in the first place - it needs some way to strip out or transform any TypeScript-specific code so that you can run it. Most TypeScript-specific code gets erased away, and likewise, here our type annotations were completely erased.

Remember: Type annotations never change the runtime behavior of your program.

### Downleveling

One other difference from the above was that our template string was rewritten from `js`Hello ${person}, today is ${date.toDateString()}!`;` to

`js"Hello ".concat(person, ", today is ").concat(date.toDateString(), "!!");`

Why did this happen?

Template strings are a feature from a version of ECMAScript called ECMAScript 2015 (a.k.a. ECMAScript 6, ES2015, ES6, etc. - don’t ask).

TypeScript has the ability to rewrite code from newer versions of ECMAScript to older ones such as ECMAScript 3 or ECMAScript 5 (a.k.a. ES3 and ES5).

This process of moving from a newer or “higher” version of ECMAScript down to an older or “lower” one is sometimes called downleveling.

By default TypeScript targets ES3, an extremely old version of ECMAScript.

We could have chosen something a little bit more recent by using the target option.

Running with `--target es2015` changes TypeScript to target ECMAScript 2015, meaning code should be able to run wherever ECMAScript 2015 is supported.

So running `tsc --target es2015 hello.ts` gives us the following output:

```
jsfunction greet(person, date) { console.log(`Hello ${person}, today is  
${date.toDateString()}!`);}greet("Maddison", new Date());
```

While the default target is ES3, the great majority of current browsers support ES2015. Most developers can therefore safely specify ES2015 or above as a target, unless compatibility with certain ancient browsers is important.

### Strictness

Different users come to TypeScript looking for different things in a type-checker.

Some people are looking for a more loose opt-in experience which can help validate only some parts of their program, and still have decent tooling.

This is the default experience with TypeScript, where types are optional, inference takes the most lenient types, and there’s no checking for potentially null/undefined values.

Much like how tsc emits in the face of errors, these defaults are put in place to stay out of your way.



If you're migrating existing JavaScript, that might be a desirable first step. In contrast, a lot of users prefer to have TypeScript validate as much as it can straight away, and that's why the language provides strictness settings as well. These strictness settings turn static type-checking from a switch (either your code is checked or not) into something closer to a dial. The further you turn this dial up, the more TypeScript will check for you. This can require a little extra work, but generally speaking it pays for itself in the long run, and enables more thorough checks and more accurate tooling. When possible, a new codebase should always turn these strictness checks on. TypeScript has several type-checking strictness flags that can be turned on or off, and all of our examples will be written with all of them enabled unless otherwise stated. The strict flag in the CLI, or "strict": true in a tsconfig.json toggles them all on simultaneously, but we can opt out of them individually. The two biggest ones you should know about are `noImplicitAny` and `strictNullChecks`.

### `noImplicitAny`

Recall that in some places, TypeScript doesn't try to infer types for us and instead falls back to the most lenient type: `any`. This isn't the worst thing that can happen - after all, falling back to `any` is just the plain JavaScript experience anyway. However, using `any` often defeats the purpose of using TypeScript in the first place. The more typed your program is, the more validation and tooling you'll get, meaning you'll run into fewer bugs as you code. Turning on the `noImplicitAny` flag will issue an error on any variables whose type is implicitly inferred as `any`.

### `strictNullChecks`

By default, values like `null` and `undefined` are assignable to any other type. This can make writing some code easier, but forgetting to handle `null` and `undefined` is the cause of countless bugs in the world - some consider it a billion dollar mistake! The `strictNullChecks` flag makes handling `null` and `undefined` more explicit, and spares us from worrying about whether we forgot to handle `null` and `undefined`. In this chapter, we'll cover some of the most common types of values you'll find in JavaScript code, and explain the corresponding ways to describe those types in TypeScript. This isn't an exhaustive list, and future chapters will describe more ways to name and use other types. Types can also appear in many more places than just type annotations. As we learn about the types themselves, we'll also learn about the places where we can refer to these types to form new constructs. We'll start by reviewing the most basic and common types you might encounter when writing JavaScript or TypeScript code. These will later form the core building blocks of more complex types. The primitives: `string`, `number`, and `boolean`. JavaScript has three very commonly used primitives: `string`, `number`, and `boolean`. Each has a corresponding type in TypeScript. As you might expect, these are the same names you'd see if you used the JavaScript `typeof` operator on a value of those types:

string represents string values like "Hello, world"  
number is for numbers like 42. JavaScript does not have a special runtime value for integers, so there's no equivalent to int or float - everything is simply number  
boolean is for the two values true and false

The type names String, Number, and Boolean (starting with capital letters) are legal, but refer to some special built-in types that will very rarely appear in your code. Always use string, number, or boolean for types.

### Arrays

To specify the type of an array like [1, 2, 3], you can use the syntax `number[]`; this syntax works for any type (e.g. `string[]` is an array of strings, and so on). You may also see this written as `Array<number>`, which means the same thing. We'll learn more about the syntax `T<U>` when we cover generics.

Note that `[number]` is a different thing; refer to the section on Tuples.

### any

TypeScript also has a special type, `any`, that you can use whenever you don't want a particular value to cause typechecking errors.

When a value is of type `any`, you can access any properties of it (which will in turn be of type `any`), call it like a function, assign it to (or from) a value of any type, or pretty much anything else that's syntactically legal:

```
tslet obj: any = { x: 0 };// None of the following lines of code will throw compiler errors.// Using `any` disables all further type checking, and it is assumed // you know the environment better than TypeScript.obj.foo();obj();obj.bar = 100;obj = "hello";const n: number = obj;Try
```

The `any` type is useful when you don't want to write out a long type just to convince TypeScript that a particular line of code is okay.

### noImplicitAny

When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to `any`.

You usually want to avoid this, though, because `any` isn't type-checked.

Use the compiler flag `noImplicitAny` to flag any implicit `any` as an error.

### Type Annotations on Variables

When you declare a variable using `const`, `var`, or `let`, you can optionally add a type annotation to explicitly specify the type of the variable:

```
tslet myName: string = "Alice";Try
```

TypeScript doesn't use "types on the left"-style declarations like `int x = 0`;

Type annotations will always go after the thing being typed.

In most cases, though, this isn't needed.

Wherever possible, TypeScript tries to automatically infer the types in your code.

For example, the type of a variable is inferred based on the type of its initializer:  
ts// No type annotation needed -- 'myName' inferred as type 'string'  
let myName = "Alice";  
Try

For the most part you don't need to explicitly learn the rules of inference.

If you're starting out, try using fewer type annotations than you think - you might be surprised how few you need for TypeScript to fully understand what's going on.

## Functions

Functions are the primary means of passing data around in JavaScript.

TypeScript allows you to specify the types of both the input and output values of functions.

### Parameter Type Annotations

When you declare a function, you can add type annotations after each parameter to declare what types of parameters the function accepts.

Parameter type annotations go after the parameter name:

```
ts// Parameter type annotation
function greet(name: string) { console.log("Hello, " +
name.toUpperCase() + "!!"); }
Try
```

When a parameter has a type annotation, arguments to that function will be checked:

```
ts// Would be a runtime error if executed!
greet(42);
Argument of type 'number' is not assignable to parameter of type 'string'.
2345
Argument of type 'number' is not assignable to parameter of type 'string'.
Try
```

Even if you don't have type annotations on your parameters, TypeScript will still check that you passed the right number of arguments.

### Return Type Annotations

You can also add return type annotations.

Return type annotations appear after the parameter list:

```
tsfunction getFavoriteNumber(): number { return 26; }
Try
```

Much like variable type annotations, you usually don't need a return type annotation because TypeScript will infer the function's return type based on its return statements.

The type annotation in the above example doesn't change anything.

Some codebases will explicitly specify a return type for documentation purposes, to prevent accidental changes, or just for personal preference.

### Anonymous Functions

Anonymous functions are a little bit different from function declarations.

When a function appears in a place where TypeScript can determine how it's going to be called, the parameters of that function are automatically given types.

Here's an example:

```
tsconst names = ["Alice", "Bob", "Eve"]; // Contextual typing for function - parameter s
inferred to have type string
names.forEach(function (s)
{ console.log(s.toUpperCase()); }); // Contextual typing also applies to arrow
functions
names.forEach((s) => { console.log(s.toUpperCase()); });
Try
```

Even though the parameter `s` didn't have a type annotation, TypeScript used the types of the `forEach` function, along with the inferred type of the array, to determine the type `s` will have.

This process is called contextual typing because the context that the function occurred

within informs what type it should have.

Similar to the inference rules, you don't need to explicitly learn how this happens, but understanding that it does happen can help you notice when type annotations aren't needed.

Later, we'll see more examples of how the context that a value occurs in can affect its type.

## Object Types

Apart from primitives, the most common sort of type you'll encounter is an object type. This refers to any JavaScript value with properties, which is almost all of them!

To define an object type, we simply list its properties and their types.

For example, here's a function that takes a point-like object:

```
ts// The parameter's type annotation is an object typefunction printCoord(pt: { x: number; y: number }) { console.log("The coordinate's x value is " + pt.x); console.log("The coordinate's y value is " + pt.y);}printCoord({ x: 3, y: 7 });Try
```

Here, we annotated the parameter with a type with two properties - x and y - which are both of type number.

You can use `,` or `;` to separate the properties, and the last separator is optional either way.

The type part of each property is also optional.

If you don't specify a type, it will be assumed to be any.

## Optional Properties

Object types can also specify that some or all of their properties are optional.

To do this, add a `?` after the property name:

```
tsfunction printName(obj: { first: string; last?: string }) { // ...}// Both OKprintName({ first: "Bob" });printName({ first: "Alice", last: "Alisson" });Try
```

In JavaScript, if you access a property that doesn't exist, you'll get the value undefined rather than a runtime error.

Because of this, when you read from an optional property, you'll have to check for undefined before using it.

```
tsfunction printName(obj: { first: string; last?: string }) { // Error - might crash if 'obj.last' wasn't provided! console.log(obj.last.toUpperCase());obj.last' is possibly 'undefined'.18048'obj.last' is possibly 'undefined'. if (obj.last !== undefined) { // OK console.log(obj.last.toUpperCase()); } // A safe alternative using modern JavaScript syntax: console.log(obj.last?.toUpperCase());}Try
```

## Union Types

TypeScript's type system allows you to build new types out of existing ones using a large variety of operators.

Now that we know how to write a few types, it's time to start combining them in interesting ways.

### Defining a Union Type

The first way to combine types you might see is a union type.

A union type is a type formed from two or more other types, representing values that may be any one of those types.

We refer to each of these types as the union's members.

Let's write a function that can operate on strings or numbers:

```
tsfunction printId(id: number | string) { console.log("Your ID is: " + id);}//
```

```
OKprintId(101); // OKprintId("202"); // ErrorprintId({ myID: 22342 });
```

Argument of type '{ myID: number; }' is not assignable to parameter of type 'string | number'.  
2345Argument of type '{ myID: number; }' is not assignable to parameter of type 'string | number'.  
Try

### Working with Union Types

It's easy to provide a value matching a union type - simply provide a type matching any of the union's members.

If you have a value of a union type, how do you work with it?

TypeScript will only allow an operation if it is valid for every member of the union.

For example, if you have the union `string | number`, you can't use methods that are only available on `string`:

```
tsfunction printId(id: number | string) { console.log(id.toUpperCase());}
```

Property 'toUpperCase' does not exist on type 'string | number'.

Property 'toUpperCase' does not exist on type 'number'.  
2339Property 'toUpperCase' does not exist on type 'string | number'.

Property 'toUpperCase' does not exist on type 'number'.  
}Try

The solution is to narrow the union with code, the same as you would in JavaScript without type annotations.

Narrowing occurs when TypeScript can deduce a more specific type for a value based on the structure of the code.

For example, TypeScript knows that only a `string` value will have a `typeof` value `"string"`:

```
tsfunction printId(id: number | string) { if (typeof id === "string") { // In this branch, id is of type 'string' console.log(id.toUpperCase()); } else { // Here, id is of type 'number' console.log(id); }}
```

Try

Another example is to use a function like `Array.isArray`:

```
tsfunction welcomePeople(x: string[] | string) { if (Array.isArray(x)) { // Here: 'x' is 'string[]' console.log("Hello, " + x.join(" and ")); } else { // Here: 'x' is 'string' console.log("Welcome lone traveler " + x); }}
```

Try

Notice that in the `else` branch, we don't need to do anything special - if `x` wasn't a `string[]`, then it must have been a `string`.

Sometimes you'll have a union where all the members have something in common.

For example, both arrays and strings have a `slice` method.

If every member in a union has a property in common, you can use that property without narrowing:

```
ts// Return type is inferred as number[] | stringfunction getFirstThree(x: number[] | string) { return x.slice(0, 3);}
```

Try

It might be confusing that a union of types appears to have the intersection of those types' properties.

This is not an accident - the name union comes from type theory.

The union `number | string` is composed by taking the union of the values from each type.

Notice that given two sets with corresponding facts about each set, only the intersection of those facts applies to the union of the sets themselves.

For example, if we had a room of tall people wearing hats, and another room of Spanish speakers wearing hats, after combining those rooms, the only thing we know

about every person is that they must be wearing a hat.

## Type Aliases

We've been using object types and union types by writing them directly in type annotations.

This is convenient, but it's common to want to use the same type more than once and refer to it by a single name.

A type alias is exactly that - a name for any type.

The syntax for a type alias is:

```
tstype Point = { x: number; y: number; }; // Exactly the same as the earlier example
function printCoord(pt: Point) { console.log("The coordinate's x value is " + pt.x); console.log("The coordinate's y value is " + pt.y); }
printCoord({ x: 100, y: 100 });
try { } catch { }
You can actually use a type alias to give a name to any type at all, not just an object type.
```

For example, a type alias can name a union type:

```
tstype ID = number | string;
try { } catch { }
```

Note that aliases are only aliases - you cannot use type aliases to create different/distinct "versions" of the same type.

When you use the alias, it's exactly as if you had written the aliased type.

In other words, this code might look illegal, but is OK according to TypeScript because both types are aliases for the same type:

```
tstype UserInputSanitizedString = string;
function sanitizeInput(str: string): UserInputSanitizedString { return sanitize(str); }
// Create a sanitized input
let userInput = sanitizeInput(getInput());
// Can still be re-assigned with a string though
userInput = "new input";
try { } catch { }
```

## Interfaces

An interface declaration is another way to name an object type:

```
interface Point { x: number; y: number; }
function printCoord(pt: Point) { console.log("The coordinate's x value is " + pt.x); console.log("The coordinate's y value is " + pt.y); }
printCoord({ x: 100, y: 100 });
try { } catch { }
```

Just like when we used a type alias above, the example works just as if we had used an anonymous object type.

TypeScript is only concerned with the structure of the value we passed to printCoord - it only cares that it has the expected properties.

Being concerned only with the structure and capabilities of types is why we call TypeScript a structurally typed type system.

## Differences Between Type Aliases and Interfaces

Type aliases and interfaces are very similar, and in many cases you can choose between them freely.

Almost all features of an interface are available in type, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

## Interface

## Type

Extending an interface

```
interface Animal {  
  name: string;  
}  
interface Bear extends Animal {  
  honey: boolean;  
}  
const bear = getBear();  
bear.name;  
bear.honey;
```

Extending a type via intersections

```
type Animal = {  
  name: string;  
}  
type Bear = Animal & {  
  honey: boolean;  
}  
const bear = getBear();  
bear.name;  
bear.honey;
```

Adding new fields to an existing interface

```
interface Window {  
  title: string;  
}  
interface Window {  
  ts: TypeScriptAPI;  
}  
const src = 'const a = "Hello World";  
window.ts.transpileModule(src, {});
```

A type cannot be changed after being created

```
type Window = {
```

```

    title: string;
}
type Window = {
  ts: TypeScriptAPI;
}
// Error: Duplicate identifier 'Window'.

```

You'll learn more about these concepts in later chapters, so don't worry if you don't understand all of these right away.

Prior to TypeScript version 4.2, type alias names may appear in error messages, sometimes in place of the equivalent anonymous type (which may or may not be desirable). Interfaces will always be named in error messages.

Type aliases may not participate in declaration merging, but interfaces can.

Interfaces may only be used to declare the shapes of objects, not rename primitives.

Interface names will always appear in their original form in error messages, but only when they are used by name.

For the most part, you can choose based on personal preference, and TypeScript will tell you if it needs something to be the other kind of declaration. If you would like a heuristic, use interface until you need to use features from type.

### Type Assertions

Sometimes you will have information about the type of a value that TypeScript can't know about.

For example, if you're using `document.getElementById`, TypeScript only knows that this will return some kind of `HTMLElement`, but you might know that your page will always have an `HTMLCanvasElement` with a given ID.

In this situation, you can use a type assertion to specify a more specific type:

```

tsconst myCanvas = document.getElementById("main_canvas") as
HTMLCanvasElement;

```

Like a type annotation, type assertions are removed by the compiler and won't affect the runtime behavior of your code.

You can also use the angle-bracket syntax (except if the code is in a `.tsx` file), which is equivalent:

```

tsconst myCanvas =
<HTMLCanvasElement>document.getElementById("main_canvas");

```

Reminder: Because type assertions are removed at compile-time, there is no runtime checking associated with a type assertion.

There won't be an exception or null generated if the type assertion is wrong.



TypeScript only allows type assertions which convert to a more specific or less specific version of a type.

This rule prevents “impossible” coercions like:

```
tsconst x = "hello" as number;
```

Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.<sup>2352</sup>

```
Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.Try
```

Sometimes this rule can be too conservative and will disallow more complex coercions that might be valid.

If this happens, you can use two assertions, first to any (or unknown, which we'll introduce later), then to the desired type:

```
tsconst a = (expr as any) as T;
```

Literal Types

In addition to the general types string and number, we can refer to specific strings and numbers in type positions.

One way to think about this is to consider how JavaScript comes with different ways to declare a variable. Both var and let allow for changing what is held inside the variable, and const does not. This is reflected in how TypeScript creates types for literals.

```
tslet changingString = "Hello World";changingString = "Olá Mundo";// Because `changingString` can represent any possible string, that// is how TypeScript describes it in the type systemchangingString;    let changingString: string const constantString = "Hello World";// Because `constantString` can only represent 1 possible string, it// has a literal type representationconstantString;    const constantString: "Hello World"Try
```

By themselves, literal types aren't very valuable:

```
tslet x: "hello" = "hello";// OKx = "hello";// ...x = "howdy";Type ""howdy"" is not assignable to type ""hello"".2322Type ""howdy"" is not assignable to type ""hello"".Try
```

It's not much use to have a variable that can only have one value!

But by combining literals into unions, you can express a much more useful concept - for example, functions that only accept a certain set of known values:

```
tsfunction printText(s: string, alignment: "left" | "right" | "center") { // ...}printText("Hello, world", "left");printText("G'day, mate", "centre");Argument of type ""centre"" is not assignable to parameter of type ""left" | "right" | "center"".2345Argument of type ""centre"" is not assignable to parameter of type ""left" | "right" | "center"".Try
```

Numeric literal types work the same way:

```
tsfunction compare(a: string, b: string): -1 | 0 | 1 { return a === b ? 0 : a > b ? 1 : -1;}Try
```

Of course, you can combine these with non-literal types:

```
tsinterface Options { width: number;}function configure(x: Options | "auto") { // ...}configure({ width: 100 });configure("auto");configure("automatic");Argument of type ""automatic"" is not assignable to parameter of type 'Options | "auto"'.2345Argument of type ""automatic"" is not assignable to parameter of type 'Options | "auto"'.Try
```

There's one more kind of literal type: boolean literals.

There are only two boolean literal types, and as you might guess, they are the types true and false.

The type boolean itself is actually just an alias for the union true | false.

Literal Inference

When you initialize a variable with an object, TypeScript assumes that the properties of that object might change values later.

For example, if you wrote code like this:

```
tsconst obj = { counter: 0 };if (someCondition) { obj.counter = 1;}Try
```

TypeScript doesn't assume the assignment of 1 to a field which previously had 0 is an error.

Another way of saying this is that `obj.counter` must have the type `number`, not `0`, because types are used to determine both reading and writing behavior.

The same applies to strings:

```
tsdeclare function handleRequest(url: string, method: "GET" | "POST"): void; const req = { url: "https://example.com", method: "GET" };handleRequest(req.url, req.method);Argument of type 'string' is not assignable to parameter of type '"GET" | "POST"'.2345Argument of type 'string' is not assignable to parameter of type '"GET" | "POST"'.Try
```

In the above example `req.method` is inferred to be `string`, not `"GET"`. Because code can be evaluated between the creation of `req` and the call of `handleRequest` which could assign a new string like `"GUESS"` to `req.method`, TypeScript considers this code to have an error.

There are two ways to work around this.

You can change the inference by adding a type assertion in either location:

```
ts// Change 1:const req = { url: "https://example.com", method: "GET" as "GET" };// Change 2handleRequest(req.url, req.method as "GET");Try
```

Change 1 means "I intend for `req.method` to always have the literal type `"GET"`", preventing the possible assignment of `"GUESS"` to that field after.

Change 2 means "I know for other reasons that `req.method` has the value `"GET"`".

You can use `as const` to convert the entire object to be type literals:

```
tsconst req = { url: "https://example.com", method: "GET" } as const;handleRequest(req.url, req.method);Try
```

The `as const` suffix acts like `const` but for the type system, ensuring that all properties are assigned the literal type instead of a more general version like `string` or `number`.  
`null` and `undefined`

JavaScript has two primitive values used to signal absent or uninitialized value: `null` and `undefined`.

TypeScript has two corresponding types by the same names. How these types behave depends on whether you have the `strictNullChecks` option on.

`strictNullChecks` off

With `strictNullChecks` off, values that might be `null` or `undefined` can still be accessed normally, and the values `null` and `undefined` can be assigned to a property of any type.

This is similar to how languages without null checks (e.g. C#, Java) behave.

The lack of checking for these values tends to be a major source of bugs; we always

recommend people turn `strictNullChecks` on if it's practical to do so in their codebase.

With `strictNullChecks` on, when a value is null or undefined, you will need to test for those values before using methods or properties on that value.

Just like checking for undefined before using an optional property, we can use narrowing to check for values that might be null:

```
tsfunction doSomething(x: string | null) { if (x === null) { // do nothing } else  
{ console.log("Hello, " + x.toUpperCase()); } }Try
```

Non-null Assertion Operator (Postfix !)

TypeScript also has a special syntax for removing null and undefined from a type without doing any explicit checking.

Writing `!` after any expression is effectively a type assertion that the value isn't null or undefined:

```
tsfunction liveDangerously(x?: number | null) { // No error console.log(x!.toFixed()); }Try
```

Just like other type assertions, this doesn't change the runtime behavior of your code, so it's important to only use `!` when you know that the value can't be null or undefined.

Enums

Enums are a feature added to JavaScript by TypeScript which allows for describing a value which could be one of a set of possible named constants. Unlike most TypeScript features, this is not a type-level addition to JavaScript but something added to the language and runtime. Because of this, it's a feature which you should know exists, but maybe hold off on using unless you are sure. You can read more about enums in the [Enum reference page](#).

Less Common Primitives

It's worth mentioning the rest of the primitives in JavaScript which are represented in the type system.

Though we will not go into depth here.

bigint

From ES2020 onwards, there is a primitive in JavaScript used for very large integers, `BigInt`:

```
ts// Creating a bigint via the BigInt functionconst oneHundred: bigint = BigInt(100); // Creating a BigInt via the literal syntaxconst anotherHundred: bigint = 100n;Try
```

You can learn more about `BigInt` in the [TypeScript 3.2 release notes](#).

symbol

There is a primitive in JavaScript used to create a globally unique reference via the function `Symbol()`:

```
tsconst firstName = Symbol("name");const secondName = Symbol("name"); if  
(firstName === secondName) {This comparison appears to be unintentional because  
the types 'typeof firstName' and 'typeof secondName' have no overlap.2367This  
comparison appears to be unintentional because the types 'typeof firstName' and  
'typeof secondName' have no overlap. // Can't ever happen}Try
```

You can learn more about them in [Symbols reference page](#).

Imagine we have a function called `padLeft`.

```
tsfunction padLeft(padding: number | string, input: string): string { throw new Error("Not  
implemented yet!"); }Try
```

If padding is a number, it will treat that as the number of spaces we want to prepend to input.

If padding is a string, it should just prepend padding to input.

Let's try to implement the logic for when padLeft is passed a number for padding.

```
tsfunction padLeft(padding: number | string, input: string) { return " ".repeat(padding) + input;}
```

Argument of type 'string | number' is not assignable to parameter of type 'number'.

Type 'string' is not assignable to type 'number'.  
Type 'string | number' is not assignable to parameter of type 'number'.

Type 'string' is not assignable to type 'number'.}

Uh-oh, we're getting an error on padding.

TypeScript is warning us that we're passing a value with type number | string to the repeat function, which only accepts a number, and it's right.

In other words, we haven't explicitly checked if padding is a number first, nor are we handling the case where it's a string, so let's do exactly that.

```
tsfunction padLeft(padding: number | string, input: string) { if (typeof padding === "number") { return " ".repeat(padding) + input; } return padding + input; }
```

If this mostly looks like uninteresting JavaScript code, that's sort of the point.

Apart from the annotations we put in place, this TypeScript code looks like JavaScript.

The idea is that TypeScript's type system aims to make it as easy as possible to write typical JavaScript code without bending over backwards to get type safety.

While it might not look like much, there's actually a lot going on under the covers here.

Much like how TypeScript analyzes runtime values using static types, it overlays type analysis on JavaScript's runtime control flow constructs like if/else, conditional ternaries, loops, truthiness checks, etc., which can all affect those types.

Within our if check, TypeScript sees typeof padding === "number" and understands that as a special form of code called a type guard.

TypeScript follows possible paths of execution that our programs can take to analyze the most specific possible type of a value at a given position.

It looks at these special checks (called type guards) and assignments, and the process of refining types to more specific types than declared is called narrowing.

In many editors we can observe these types as they change, and we'll even do so in our examples.

```
tsfunction padLeft(padding: number | string, input: string) { if (typeof padding === "number") { return " ".repeat(padding) + input; } else { return padding + input; } }
```

There are a couple of different constructs TypeScript understands for narrowing.

typeof type guards

As we've seen, JavaScript supports a typeof operator which can give very basic information about the type of values we have at runtime.

TypeScript expects this to return a certain set of strings:

"string"

"number"

"bigint"

"boolean"

"symbol"

"undefined"

"object"

"function"

Like we saw with `padLeft`, this operator comes up pretty often in a number of JavaScript libraries, and TypeScript can understand it to narrow types in different branches.

In TypeScript, checking against the value returned by `typeof` is a type guard.

Because TypeScript encodes how `typeof` operates on different values, it knows about some of its quirks in JavaScript.

For example, notice that in the list above, `typeof` doesn't return the string `null`.

Check out the following example:

```
tsfunction printAll(strs: string | string[] | null) { if (typeof strs === "object") { for (const s of strs) { 'strs' is possibly 'null'.18047'strs' is possibly 'null'. console.log(s); } } else if (typeof strs === "string") { console.log(strs); } else { // do nothing }}Try
```

In the `printAll` function, we try to check if `strs` is an object to see if it's an array type (now might be a good time to reinforce that arrays are object types in JavaScript).

But it turns out that in JavaScript, `typeof null` is actually `"object"`!

This is one of those unfortunate accidents of history.

Users with enough experience might not be surprised, but not everyone has run into this in JavaScript; luckily, TypeScript lets us know that `strs` was only narrowed down to `string[] | null` instead of just `string[]`.

This might be a good segue into what we'll call "truthiness" checking.

Truthiness narrowing

Truthiness might not be a word you'll find in the dictionary, but it's very much something you'll hear about in JavaScript.

In JavaScript, we can use any expression in conditionals, `&&`s, `||`s, if statements, Boolean negations (`!`), and more.

As an example, if statements don't expect their condition to always have the type `boolean`.

```
tsfunction getUsersOnlineMessage(numUsersOnline: number) { if (numUsersOnline) { return `There are ${numUsersOnline} online now!`; } return "Nobody's here. :("; }Try
```

In JavaScript, constructs like `if` first "coerce" their conditions to booleans to make sense of them, and then choose their branches depending on whether the result is `true` or `false`.

Values like

`0`

`NaN`

`""` (the empty string)

`0n` (the bigint version of zero)

`null`

`undefined`

all coerce to `false`, and other values get coerced to `true`.

You can always coerce values to booleans by running them through the `Boolean` function, or by using the shorter double-Boolean negation. (The latter has the advantage that TypeScript infers a narrow literal boolean type `true`, while inferring the first as type `boolean`.)

```
ts// both of these result in 'true'Boolean("hello"); // type: boolean, value: true!!"world"; //
type: true, value: trueTry
```

It's fairly popular to leverage this behavior, especially for guarding against values like null or undefined.

As an example, let's try using it for our printAll function.

```
tsfunction printAll(strs: string | string[] | null) { if (strs && typeof strs === "object") { for
(const s of strs) { console.log(s); } } else if (typeof strs === "string")
{ console.log(strs); }}Try
```

You'll notice that we've gotten rid of the error above by checking if strs is truthy.

This at least prevents us from dreaded errors when we run our code like:

```
txtTypeError: null is not iterable
```

Keep in mind though that truthiness checking on primitives can often be error prone.

As an example, consider a different attempt at writing printAll

```
tsfunction printAll(strs: string | string[] | null) { // !!!!!!!!!!!!!!! // DON'T DO THIS! //
KEEP READING // !!!!!!!!!!!!!!! if (strs) { if (typeof strs === "object") { for (const s
of strs) { console.log(s); } } else if (typeof strs === "string")
{ console.log(strs); } }}Try
```

We wrapped the entire body of the function in a truthy check, but this has a subtle downside: we may no longer be handling the empty string case correctly.

TypeScript doesn't hurt us here at all, but this behavior is worth noting if you're less familiar with JavaScript.

TypeScript can often help you catch bugs early on, but if you choose to do nothing with a value, there's only so much that it can do without being overly prescriptive.

If you want, you can make sure you handle situations like these with a linter.

One last word on narrowing by truthiness is that Boolean negations with ! filter out from negated branches.

```
tsfunction multiplyAll( values: number[] | undefined, factor: number): number[] |
undefined { if (!values) { return values; } else { return values.map((x) => x *
factor); }}Try
```

Equality narrowing

TypeScript also uses switch statements and equality checks like ===, !==, ==, and != to narrow types.

For example:

```
tsfunction example(x: string | number, y: string | boolean) { if (x === y) { // We can
now call any 'string' method on 'x' or 'y'. x.toUpperCase(); (method)
String.toUpperCase(): string y.toLowerCase(); (method) String.toLowerCase():
string } else { console.log(x); (parameter) x: string | number
console.log(y); (parameter) y: string | boolean }}Try
```

When we checked that x and y are both equal in the above example, TypeScript knew their types also had to be equal.

Since string is the only common type that both x and y could take on, TypeScript knows that x and y must be a string in the first branch.

Checking against specific literal values (as opposed to variables) works also.

In our section about truthiness narrowing, we wrote a printAll function which was error-prone because it accidentally didn't handle empty strings properly.

Instead we could have done a specific check to block out nulls, and TypeScript still

correctly removes null from the type of strs.

```
tsfunction printAll(strs: string | string[] | null) { if (strs !== null) { if (typeof strs === "object") { for (const s of strs) { (parameter) strs: string[] console.log(s); } } else if (typeof strs === "string") { console.log(strs); (parameter) strs: string } }}Try
```

JavaScript's looser equality checks with == and != also get narrowed correctly.

If you're unfamiliar, checking whether something == null actually not only checks whether it is specifically the value null - it also checks whether it's potentially undefined. The same applies to == undefined: it checks whether a value is either null or undefined.

```
tsinterface Container { value: number | null | undefined;} function multiplyValue(container: Container, factor: number) { // Remove both 'null' and 'undefined' from the type. if (container.value !== null) { console.log(container.value); (property) Container.value: number // Now we can safely multiply 'container.value'. container.value *= factor; }}Try
```

The in operator narrowing

JavaScript has an operator for determining if an object or its prototype chain has a property with a name: the in operator.

TypeScript takes this into account as a way to narrow down potential types.

For example, with the code: "value" in x. where "value" is a string literal and x is a union type.

The "true" branch narrows x's types which have either an optional or required property value, and the "false" branch narrows to types which have an optional or missing property value.

```
tstype Fish = { swim: () => void };type Bird = { fly: () => void }; function move(animal: Fish | Bird) { if ("swim" in animal) { return animal.swim(); } return animal.fly();}Try
```

To reiterate, optional properties will exist in both sides for narrowing. For example, a human could both swim and fly (with the right equipment) and thus should show up in both sides of the in check:

```
tstype Fish = { swim: () => void };type Bird = { fly: () => void };type Human = { swim?: () => void; fly?: () => void }; function move(animal: Fish | Bird | Human) { if ("swim" in animal) { animal; (parameter) animal: Fish | Human } else { animal; (parameter) animal: Bird | Human }}Try
```

instanceof narrowing

JavaScript has an operator for checking whether or not a value is an "instance" of another value.

More specifically, in JavaScript x instanceof Foo checks whether the prototype chain of x contains Foo.prototype.

While we won't dive deep here, and you'll see more of this when we get into classes, they can still be useful for most values that can be constructed with new.

As you might have guessed, instanceof is also a type guard, and TypeScript narrows in branches guarded by instanceof.

```
tsfunction logValue(x: Date | string) { if (x instanceof Date) { console.log(x.toUTCString()); (parameter) x: Date } else { console.log(x.toUpperCase()); (parameter) x: string }}Try
```

## Assignments

As we mentioned earlier, when we assign to any variable, TypeScript looks at the right side of the assignment and narrows the left side appropriately.

```
tslet x = Math.random() < 0.5 ? 10 : "hello world!"; let x: string | number; x = 1; console.log(x); let x: number; x = "goodbye!"; console.log(x); let x: string; x = 10; console.log(x);
```

Notice that each of these assignments is valid.

Even though the observed type of `x` changed to `number` after our first assignment, we were still able to assign a string to `x`.

This is because the declared type of `x` - the type that `x` started with - is `string | number`, and assignability is always checked against the declared type.

If we'd assigned a `boolean` to `x`, we'd have seen an error since that wasn't part of the declared type.

```
tslet x = Math.random() < 0.5 ? 10 : "hello world!"; let x: string | number; x = 1; console.log(x); let x: number; x = true; // Type 'boolean' is not assignable to type 'string | number'. console.log(x); let x: string | number; x = 10; console.log(x);
```

## Control flow analysis

Up until this point, we've gone through some basic examples of how TypeScript narrows within specific branches.

But there's a bit more going on than just walking up from every variable and looking for type guards in `ifs`, `whiles`, `conditionals`, etc.

For example

```
tsfunction padLeft(padding: number | string, input: string) { if (typeof padding === "number") { return " ".repeat(padding) + input; } return padding + input; } Try padLeft returns from within its first if block.
```

TypeScript was able to analyze this code and see that the rest of the body (`return padding + input;`) is unreachable in the case where `padding` is a `number`.

As a result, it was able to remove `number` from the type of `padding` (narrowing from `string | number` to `string`) for the rest of the function.

This analysis of code based on reachability is called control flow analysis, and TypeScript uses this flow analysis to narrow types as it encounters type guards and assignments.

When a variable is analyzed, control flow can split off and re-merge over and over again, and that variable can be observed to have a different type at each point.

```
tsfunction example() { let x: string | number | boolean; x = Math.random() < 0.5; console.log(x); let x: boolean; if (Math.random() < 0.5) { x = "hello"; console.log(x); } else { x = 100; console.log(x); } let x: number; x = true; console.log(x); return x; } Try
```

## Using type predicates

We've worked with existing JavaScript constructs to handle narrowing so far, however sometimes you want more direct control over how types change throughout your code.

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

```
tsfunction isFish(pet: Fish | Bird): pet is Fish { return (pet as Fish).swim !== undefined; } Try
```

`pet is Fish` is our type predicate in this example.



A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

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

```
ts// Both calls to 'swim' and 'fly' are now okay.let pet = getSmallPet(); if (isFish(pet))
{ pet.swim();} else { pet.fly();}Try
```

Notice that TypeScript not only knows that `pet` is a `Fish` in the `if` branch;

it also knows that in the `else` branch, you don't have a `Fish`, so you must have a `Bird`.

You may use the type guard `isFish` to filter an array of `Fish | Bird` and obtain an array of `Fish`:

```
tsconst zoo: (Fish | Bird)[] = [getSmallPet(), getSmallPet(), getSmallPet()];const
underWater1: Fish[] = zoo.filter(isFish);// or, equivalentlyconst underWater2: Fish[] =
zoo.filter(isFish) as Fish[]; // The predicate may need repeating for more complex
examplesconst underWater3: Fish[] = zoo.filter((pet): pet is Fish => { if (pet.name ===
"sharkey") return false; return isFish(pet);});Try
```

In addition, classes can use this `is Type` to narrow their type.

Assertion functions

Types can also be narrowed using Assertion functions.

Discriminated unions

Most of the examples we've looked at so far have focused around narrowing single variables with simple types like `string`, `boolean`, and `number`.

While this is common, most of the time in JavaScript we'll be dealing with slightly more complex structures.

For some motivation, let's imagine we're trying to encode shapes like circles and squares.

Circles keep track of their radiuses and squares keep track of their side lengths.

We'll use a field called `kind` to tell which shape we're dealing with.

Here's a first attempt at defining `Shape`.

```
tsinterface Shape { kind: "circle" | "square"; radius?: number; sideLength?: number;}Try
```

Notice we're using a union of string literal types: `"circle"` and `"square"` to tell us whether we should treat the shape as a circle or square respectively.

By using `"circle" | "square"` instead of `string`, we can avoid misspelling issues.

```
tsfunction handleShape(shape: Shape) { // oops! if (shape.kind === "rect") {This
comparison appears to be unintentional because the types '"circle" | "square"' and
"rect" have no overlap.2367This comparison appears to be unintentional because the
types '"circle" | "square"' and "rect" have no overlap. // ... }}Try
```

We can write a `getArea` function that applies the right logic based on if it's dealing with a circle or square.

We'll first try dealing with circles.

```
tsfunction getArea(shape: Shape) { return Math.PI * shape.radius ** 2;'shape.radius' is
possibly 'undefined'.18048'shape.radius' is possibly 'undefined'.}Try
```

Under `strictNullChecks` that gives us an error - which is appropriate since `radius` might not be defined.

But what if we perform the appropriate checks on the `kind` property?

```
tsfunction getArea(shape: Shape) { if (shape.kind === "circle") { return Math.PI *
```

```
shape.radius ** 2; 'shape.radius' is possibly 'undefined'.18048'shape.radius' is possibly 'undefined'. }}Try
```

Hmm, TypeScript still doesn't know what to do here.

We've hit a point where we know more about our values than the type checker does. We could try to use a non-null assertion (a ! after shape.radius) to say that radius is definitely present.

```
tsfunction getArea(shape: Shape) { if (shape.kind === "circle") { return Math.PI * shape.radius! ** 2; }}Try
```

But this doesn't feel ideal.

We had to shout a bit at the type-checker with those non-null assertions (!) to convince it that shape.radius was defined, but those assertions are error-prone if we start to move code around.

Additionally, outside of strictNullChecks we're able to accidentally access any of those fields anyway (since optional properties are just assumed to always be present when reading them).

We can definitely do better.

The problem with this encoding of Shape is that the type-checker doesn't have any way to know whether or not radius or sideLength are present based on the kind property.

We need to communicate what we know to the type checker.

With that in mind, let's take another swing at defining Shape.

```
tsinterface Circle { kind: "circle"; radius: number;} interface Square { kind: "square"; sideLength: number;} type Shape = Circle | Square;Try
```

Here, we've properly separated Shape out into two types with different values for the kind property, but radius and sideLength are declared as required properties in their respective types.

Let's see what happens here when we try to access the radius of a Shape.

```
tsfunction getArea(shape: Shape) { return Math.PI * shape.radius ** 2;Property 'radius' does not exist on type 'Shape'.
```

```
Property 'radius' does not exist on type 'Square'.2339Property 'radius' does not exist on type 'Shape'.
```

```
Property 'radius' does not exist on type 'Square'.}Try
```

Like with our first definition of Shape, this is still an error.

When radius was optional, we got an error (with strictNullChecks enabled) because TypeScript couldn't tell whether the property was present.

Now that Shape is a union, TypeScript is telling us that shape might be a Square, and Squares don't have radius defined on them!

Both interpretations are correct, but only the union encoding of Shape will cause an error regardless of how strictNullChecks is configured.

But what if we tried checking the kind property again?

```
tsfunction getArea(shape: Shape) { if (shape.kind === "circle") { return Math.PI * shape.radius ** 2; (parameter) shape: Circle }}Try
```

That got rid of the error!

When every type in a union contains a common property with literal types, TypeScript considers that to be a discriminated union, and can narrow out the members of the union.

In this case, kind was that common property (which is what's considered a discriminant

property of Shape).

Checking whether the kind property was "circle" got rid of every type in Shape that didn't have a kind property with the type "circle".

That narrowed shape down to the type Circle.

The same checking works with switch statements as well.

Now we can try to write our complete getArea without any pesky ! non-null assertions.

```
tsfunction getArea(shape: Shape) { switch (shape.kind) { case "circle": return  
Math.PI * shape.radius ** 2; (parameter) shape: Circle case  
"square": return shape.sideLength ** 2; (parameter) shape: Square }}Try
```

The important thing here was the encoding of Shape.

Communicating the right information to TypeScript - that Circle and Square were really two separate types with specific kind fields - was crucial.

Doing that lets us write type-safe TypeScript code that looks no different than the JavaScript we would've written otherwise.

From there, the type system was able to do the "right" thing and figure out the types in each branch of our switch statement.

As an aside, try playing around with the above example and remove some of the return keywords.

You'll see that type-checking can help avoid bugs when accidentally falling through different clauses in a switch statement.

Discriminated unions are useful for more than just talking about circles and squares.

They're good for representing any sort of messaging scheme in JavaScript, like when sending messages over the network (client/server communication), or encoding mutations in a state management framework.

The never type

When narrowing, you can reduce the options of a union to a point where you have removed all possibilities and have nothing left.

In those cases, TypeScript will use a never type to represent a state which shouldn't exist.

Exhaustiveness checking

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

For example, adding a default to our getArea function which tries to assign the shape to never will not raise an error when every possible case has been handled.

```
tsinterface Shape = Circle | Square; function getArea(shape: Shape) { switch (shape.kind)  
{ case "circle": return Math.PI * shape.radius ** 2; case "square": return  
shape.sideLength ** 2; default: const _exhaustiveCheck: never = shape; return  
_exhaustiveCheck; }}Try
```

Adding a new member to the Shape union, will cause a TypeScript error:

```
tsinterface Triangle { kind: "triangle"; sideLength: number;} type Shape = Circle |  
Square | Triangle; function getArea(shape: Shape) { switch (shape.kind) { case  
"circle": return Math.PI * shape.radius ** 2; case "square": return  
shape.sideLength ** 2; default: const _exhaustiveCheck: never = shape;Type
```

'Triangle' is not assignable to type 'never'.2322Type 'Triangle' is not assignable to type 'never'.  
return \_exhaustiveCheck; } }Try

Functions are the basic building block of any application, whether they're local functions, imported from another module, or methods on a class.

They're also values, and just like other values, TypeScript has many ways to describe how functions can be called.

Let's learn about how to write types that describe functions.

### Function Type Expressions

The simplest way to describe a function is with a function type expression.

These types are syntactically similar to arrow functions:

```
tsfunction greeter(fn: (a: string) => void) { fn("Hello, World");} function printToConsole(s: string) { console.log(s);} greeter(printToConsole);Try
```

The syntax `(a: string) => void` means “a function with one parameter, named `a`, of type `string`, that doesn't have a return value”.

Just like with function declarations, if a parameter type isn't specified, it's implicitly `any`.

Note that the parameter name is required. The function type `(string) => void` means “a function with a parameter named `string` of type `any`”!

Of course, we can use a type alias to name a function type:

```
tstype GreetFunction = (a: string) => void;function greeter(fn: GreetFunction) { // ...}Try
```

### Call Signatures

In JavaScript, functions can have properties in addition to being callable.

However, the function type expression syntax doesn't allow for declaring properties.

If we want to describe something callable with properties, we can write a call signature in an object type:

```
tstype DescribableFunction = { description: string; (someArg: number): boolean;};function doSomething(fn: DescribableFunction) { console.log(fn.description + " returned " + fn(6));} function myFunc(someArg: number) { return someArg > 3;} myFunc.description = "default description"; doSomething(myFunc);Try
```

Note that the syntax is slightly different compared to a function type expression - use `:` between the parameter list and the return type rather than `=>`.

### Construct Signatures

JavaScript functions can also be invoked with the `new` operator.

TypeScript refers to these as constructors because they usually create a new object.

You can write a construct signature by adding the `new` keyword in front of a call signature:

```
tstype SomeConstructor = { new (s: string): SomeObject;};function fn(ctor: SomeConstructor) { return new ctor("hello");}Try
```

Some objects, like JavaScript's `Date` object, can be called with or without `new`.

You can combine call and construct signatures in the same type arbitrarily:

```
tsinterface CallOrConstruct { new (s: string): Date; (n?: number): string;}Try
```

### Generic Functions

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

Let's consider for a moment a function that returns the first element of an array:

```
tsfunction firstElement(arr: any[]) { return arr[0];}Try
```

This function does its job, but unfortunately has the return type any.

It'd be better if the function returned the type of the array element.

In TypeScript, generics are used when we want to describe a correspondence between two values.

We do this by declaring a type parameter in the function signature:

```
tsfunction firstElement<Type>(arr: Type[]): Type | undefined { return arr[0];}Try
```

By adding a type parameter Type to this function and using it in two places, we've created a link between the input of the function (the array) and the output (the return value).

Now when we call it, a more specific type comes out:

```
ts// s is of type 'string'const s = firstElement(["a", "b", "c"]);// n is of type 'number'const n = firstElement([1, 2, 3]);// u is of type undefinedconst u = firstElement([]);Try
```

Inference

Note that we didn't have to specify Type in this sample.

The type was inferred - chosen automatically - by TypeScript.

We can use multiple type parameters as well.

For example, a standalone version of map would look like this:

```
tsfunction map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] { return arr.map(func);} // Parameter 'n' is of type 'string'// 'parsed' is of type 'number[]'const parsed = map(["1", "2", "3"], (n) => parseInt(n));Try
```

Note that in this example, TypeScript could infer both the type of the Input type parameter (from the given string array), as well as the Output type parameter based on the return value of the function expression (number).

Constraints

We've written some generic functions that can work on any kind of value.

Sometimes we want to relate two values, but can only operate on a certain subset of values.

In this case, we can use a constraint to limit the kinds of types that a type parameter can accept.

Let's write a function that returns the longer of two values.

To do this, we need a length property that's a number.

We constrain the type parameter to that type by writing an extends clause:

```
tsfunction longest<Type extends { length: number }>(a: Type, b: Type) { if (a.length >= b.length) { return a; } else { return b; }} // longerArray is of type 'number[]'const longerArray = longest([1, 2], [1, 2, 3]);// longerString is of type 'alice' | 'bob'const longerString = longest("alice", "bob");// Error! Numbers don't have a 'length' propertyconst notOK = longest(10, 100);Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.2345Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.Try
```

There are a few interesting things to note in this example.

We allowed TypeScript to infer the return type of longest.

Return type inference also works on generic functions.

Because we constrained Type to { length: number }, we were allowed to access the .length property of the a and b parameters.

Without the type constraint, we wouldn't be able to access those properties because

the values might have been some other type without a length property.

The types of `longerArray` and `longerString` were inferred based on the arguments.

Remember, generics are all about relating two or more values with the same type!

Finally, just as we'd like, the call to `longest(10, 100)` is rejected because the number type doesn't have a `.length` property.

### Working with Constrained Values

Here's a common error when working with generic constraints:

```
tsfunction minimumLength<Type extends { length: number }>( obj: Type, minimum: number): Type { if (obj.length >= minimum) { return obj; } else { return { length: minimum }; } }Type '{ length: number; }' is not assignable to type 'Type'.
```

'{ length: number; }' is assignable to the constraint of type 'Type', but 'Type' could be instantiated with a different subtype of constraint '{ length: number; }'.  
'{ length: number; }' is not assignable to type 'Type'.

'{ length: number; }' is assignable to the constraint of type 'Type', but 'Type' could be instantiated with a different subtype of constraint '{ length: number; }'.

It might look like this function is OK - `Type` is constrained to `{ length: number }`, and the function either returns `Type` or a value matching that constraint.

The problem is that the function promises to return the same kind of object as was passed in, not just some object matching the constraint.

If this code were legal, you could write code that definitely wouldn't work:

```
ts// 'arr' gets value { length: 6 }const arr = minimumLength([1, 2, 3], 6);// and crashes here because arrays have a 'slice' method, but not the returned object!
```

```
console.log(arr.slice(0));
```

### Specifying Type Arguments

TypeScript can usually infer the intended type arguments in a generic call, but not always.

For example, let's say you wrote a function to combine two arrays:

```
tsfunction combine<Type>(arr1: Type[], arr2: Type[]): Type[] { return arr1.concat(arr2); }
Try
```

Normally it would be an error to call this function with mismatched arrays:

```
tsconst arr = combine([1, 2, 3], ["hello"]);Type 'string' is not assignable to type 'number'.
Type 'string' is not assignable to type 'number'.
```

If you intended to do this, however, you could manually specify `Type`:

```
tsconst arr = combine<string | number>([1, 2, 3], ["hello"]);
```

### Guidelines for Writing Good Generic Functions

Writing generic functions is fun, and it can be easy to get carried away with type parameters.

Having too many type parameters or using constraints where they aren't needed can make inference less successful, frustrating callers of your function.

### Push Type Parameters Down

Here are two ways of writing a function that appear similar:

```
tsfunction firstElement1<Type>(arr: Type[]) { return arr[0]; }
function firstElement2<Type extends any[]>(arr: Type) { return arr[0]; }
// a: number (good)
const a = firstElement1([1, 2, 3]);
// b: any (bad)
const b = firstElement2([1, 2, 3]);
```

These might seem identical at first glance, but `firstElement1` is a much better way to write this function.

Its inferred return type is `Type`, but `firstElement2`'s inferred return type is `any` because TypeScript has to resolve the `arr[0]` expression using the constraint type, rather than “waiting” to resolve the element during a call.

Rule: When possible, use the type parameter itself rather than constraining it

### Use Fewer Type Parameters

Here's another pair of similar functions:

```
tsfunction filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] { return  
arr.filter(func);} function filter2<Type, Func extends (arg: Type) => boolean>( arr:  
Type[], func: Func): Type[] { return arr.filter(func);}Try
```

We've created a type parameter `Func` that doesn't relate two values.

That's always a red flag, because it means callers wanting to specify type arguments have to manually specify an extra type argument for no reason.

`Func` doesn't do anything but make the function harder to read and reason about!

Rule: Always use as few type parameters as possible

### Type Parameters Should Appear Twice

Sometimes we forget that a function might not need to be generic:

```
tsfunction greet<Str extends string>(s: Str) { console.log("Hello, " +  
s);} greet("world");Try
```

We could just as easily have written a simpler version:

```
tsfunction greet(s: string) { console.log("Hello, " + s);}Try
```

Remember, type parameters are for relating the types of multiple values.

If a type parameter is only used once in the function signature, it's not relating anything.

This includes the inferred return type; for example, if `Str` was part of the inferred return type of `greet`, it would be relating the argument and return types, so would be used twice despite appearing only once in the written code.

Rule: If a type parameter only appears in one location, strongly reconsider if you actually need it

### Optional Parameters

Functions in JavaScript often take a variable number of arguments.

For example, the `toFixed` method of `number` takes an optional digit count:

```
tsfunction f(n: number) { console.log(n.toFixed()); // 0 arguments  
console.log(n.toFixed(3)); // 1 argument}Try
```

We can model this in TypeScript by marking the parameter as optional with `?`:

```
tsfunction f(x?: number) { // ...}f(); // OKf(10); // OKTry
```

Although the parameter is specified as type `number`, the `x` parameter will actually have the type `number | undefined` because unspecified parameters in JavaScript get the value `undefined`.

You can also provide a parameter default:

```
tsfunction f(x = 10) { // ...}Try
```

Now in the body of `f`, `x` will have type `number` because any undefined argument will be

replaced with 10.

Note that when a parameter is optional, callers can always pass undefined, as this simply simulates a “missing” argument:

```
tsdeclare function f(x?: number): void; // cut// All OKf();f(10);f(undefined);Try
```

### Optional Parameters in Callbacks

Once you’ve learned about optional parameters and function type expressions, it’s very easy to make the following mistakes when writing functions that invoke callbacks:

```
tsfunction myForEach(arr: any[], callback: (arg: any, index?: number) => void) { for (let i = 0; i < arr.length; i++) { callback(arr[i], i); }}Try
```

What people usually intend when writing index? as an optional parameter is that they want both of these calls to be legal:

```
tsmyForEach([1, 2, 3], (a) => console.log(a));myForEach([1, 2, 3], (a, i) => console.log(a, i));Try
```

What this actually means is that callback might get invoked with one argument.

In other words, the function definition says that the implementation might look like this:

```
tsfunction myForEach(arr: any[], callback: (arg: any, index?: number) => void) { for (let i = 0; i < arr.length; i++) { // I don't feel like providing the index today callback(arr[i]); }}
```

Try

In turn, TypeScript will enforce this meaning and issue errors that aren’t really possible:

```
tsmyForEach([1, 2, 3], (a, i) => { console.log(i.toFixed());'i' is possibly 'undefined'.18048'i' is possibly 'undefined'.});Try
```

In JavaScript, if you call a function with more arguments than there are parameters, the extra arguments are simply ignored.

TypeScript behaves the same way.

Functions with fewer parameters (of the same types) can always take the place of functions with more parameters.

Rule: When writing a function type for a callback, never write an optional parameter unless you intend to call the function without passing that argument

### Function Overloads

Some JavaScript functions can be called in a variety of argument counts and types.

For example, you might write a function to produce a Date that takes either a timestamp (one argument) or a month/day/year specification (three arguments).

In TypeScript, we can specify a function that can be called in different ways by writing overload signatures.

To do this, write some number of function signatures (usually two or more), followed by the body of the function:

```
tsfunction makeDate(timestamp: number): Date;function makeDate(m: number, d: number, y: number): Date;function makeDate(mOrTimestamp: number, d?: number, y?: number): Date { if (d !== undefined && y !== undefined) { return new Date(y, mOrTimestamp, d); } else { return new Date(mOrTimestamp); }}const d1 = makeDate(12345678);const d2 = makeDate(5, 5, 5);const d3 = makeDate(1, 3);No overload expects 2 arguments, but overloads do exist that expect either 1 or 3 arguments.2575No overload expects 2 arguments, but overloads do exist that expect
```



either 1 or 3 arguments. Try

In this example, we wrote two overloads: one accepting one argument, and another accepting three arguments.

These first two signatures are called the overload signatures.

Then, we wrote a function implementation with a compatible signature.

Functions have an implementation signature, but this signature can't be called directly. Even though we wrote a function with two optional parameters after the required one, it can't be called with two parameters!

Overload Signatures and the Implementation Signature

This is a common source of confusion.

Often people will write code like this and not understand why there is an error:

```
tsfunction fn(x: string): void;function fn() { // ...} // Expected to be able to call with zero argumentsfn(); // Expected 1 arguments, but got 0.2554Expected 1 arguments, but got 0. Try
```

Again, the signature used to write the function body can't be "seen" from the outside.

The signature of the implementation is not visible from the outside.

When writing an overloaded function, you should always have two or more signatures above the implementation of the function.

The implementation signature must also be compatible with the overload signatures.

For example, these functions have errors because the implementation signature doesn't match the overloads in a correct way:

```
tsfunction fn(x: boolean): void; // Argument type isn't rightfunction fn(x: string): void; // This overload signature is not compatible with its implementation signature.2394This overload signature is not compatible with its implementation signature.function fn(x: boolean) {} Try
```

```
tsfunction fn(x: string): string; // Return type isn't rightfunction fn(x: number): boolean; // This overload signature is not compatible with its implementation signature.2394This overload signature is not compatible with its implementation signature.function fn(x: string | number) { return "oops"; } Try
```

Writing Good Overloads

Like generics, there are a few guidelines you should follow when using function overloads.

Following these principles will make your function easier to call, easier to understand, and easier to implement.

Let's consider a function that returns the length of a string or an array:

```
tsfunction len(s: string): number;function len(arr: any[]): number;function len(x: any) { return x.length; } Try
```

This function is fine; we can invoke it with strings or arrays.

However, we can't invoke it with a value that might be a string or an array, because TypeScript can only resolve a function call to a single overload:

```
tslen(""); // OKlen([0]); // OKlen(Math.random() > 0.5 ? "hello" : [0]); // No overload matches this call.
```

Overload 1 of 2, '(s: string): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'string'.

Type 'number[]' is not assignable to type 'string'.

Overload 2 of 2, '(arr: any[]): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'any[]'.

Type 'string' is not assignable to type 'any[]'.2769No overload matches this call.

Overload 1 of 2, '(s: string): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'string'.

Type 'number[]' is not assignable to type 'string'.

Overload 2 of 2, '(arr: any[]): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'any[]'.

Type 'string' is not assignable to type 'any[]'.Try

Because both overloads have the same argument count and same return type, we can instead write a non-overloaded version of the function:

```
tsfunction len(x: any[] | string) { return x.length;}Try
```

This is much better!

Callers can invoke this with either sort of value, and as an added bonus, we don't have to figure out a correct implementation signature.

Always prefer parameters with union types instead of overloads when possible

### Declaring this in a Function

TypeScript will infer what the `this` should be in a function via code flow analysis, for example in the following:

```
tsconst user = { id: 123, admin: false, becomeAdmin: function () { this.admin = true; } };Try
```

TypeScript understands that the function `user.becomeAdmin` has a corresponding `this` which is the outer object `user`. `this`, heh, can be enough for a lot of cases, but there are a lot of cases where you need more control over what object `this` represents. The JavaScript specification states that you cannot have a parameter called `this`, and so TypeScript uses that syntax space to let you declare the type for `this` in the function body.

```
tsinterface DB { filterUsers(filter: (this: User) => boolean): User[]; } const db = getDB();const admins = db.filterUsers(function (this: User) { return this.admin;});Try
```

This pattern is common with callback-style APIs, where another object typically controls when your function is called. Note that you need to use function and not arrow functions to get this behavior:

```
tsinterface DB { filterUsers(filter: (this: User) => boolean): User[]; } const db = getDB();const admins = db.filterUsers(() => this.admin);The containing arrow function captures the global value of 'this'.Element implicitly has an 'any' type because type 'typeof globalThis' has no index signature.70417017The containing arrow function captures the global value of 'this'.Element implicitly has an 'any' type because type 'typeof globalThis' has no index signature.Try
```

### Other Types to Know About

There are some additional types you'll want to recognize that appear often when working with function types.

Like all types, you can use them everywhere, but these are especially relevant in the context of functions.

void

void represents the return value of functions which don't return a value.

It's the inferred type any time a function doesn't have any return statements, or doesn't return any explicit value from those return statements:

```
ts// The inferred return type is voidfunction noop() { return;}Try
```

In JavaScript, a function that doesn't return any value will implicitly return the value undefined.

However, void and undefined are not the same thing in TypeScript.

There are further details at the end of this chapter.

void is not the same as undefined.

object

The special type object refers to any value that isn't a primitive (string, number, bigint, boolean, symbol, null, or undefined).

This is different from the empty object type { }, and also different from the global type Object.

It's very likely you will never use Object.

object is not Object. Always use object!

Note that in JavaScript, function values are objects: They have properties, have Object.prototype in their prototype chain, are instanceof Object, you can call Object.keys on them, and so on.

For this reason, function types are considered to be objects in TypeScript.

unknown

The unknown type represents any value.

This is similar to the any type, but is safer because it's not legal to do anything with an unknown value:

```
tsfunction f1(a: any) { a.b(); // OK}function f2(a: unknown) { a.b(); 'a' is of type 'unknown'.18046'a' is of type 'unknown'.}Try
```

This is useful when describing function types because you can describe functions that accept any value without having any values in your function body.

Conversely, you can describe a function that returns a value of unknown type:

```
tsfunction safeParse(s: string): unknown { return JSON.parse(s); } // Need to be careful with 'obj'!const obj = safeParse(someRandomString);Try
```

never

Some functions never return a value:

```
tsfunction fail(msg: string): never { throw new Error(msg); }Try
```

The never type represents values which are never observed.

In a return type, this means that the function throws an exception or terminates execution of the program.

never also appears when TypeScript determines there's nothing left in a union.

```
tsfunction fn(x: string | number) { if (typeof x === "string") { // do something } else if (typeof x === "number") { // do something else } else { x; // has type 'never!' }}Try
```

Function

The global type `Function` describes properties like `bind`, `call`, `apply`, and others present on all function values in JavaScript.

It also has the special property that values of type `Function` can always be called; these calls return `any`:

```
tsfunction doSomething(f: Function) { return f(1, 2, 3);}Try
```

This is an untyped function call and is generally best avoided because of the unsafe `any` return type.

If you need to accept an arbitrary function but don't intend to call it, the type `() => void` is generally safer.

## Rest Parameters and Arguments

Background Reading:

Rest Parameters

Spread Syntax

### Rest Parameters

In addition to using optional parameters or overloads to make functions that can accept a variety of fixed argument counts, we can also define functions that take an unbounded number of arguments using rest parameters.

A rest parameter appears after all other parameters, and uses the `...` syntax:

```
tsfunction multiply(n: number, ...m: number[]) { return m.map((x) => n * x);}// 'a' gets value [10, 20, 30, 40]const a = multiply(10, 1, 2, 3, 4);Try
```

In TypeScript, the type annotation on these parameters is implicitly `any[]` instead of `any`, and any type annotation given must be of the form `Array<T>` or `T[]`, or a tuple type (which we'll learn about later).

### Rest Arguments

Conversely, we can provide a variable number of arguments from an iterable object (for example, an array) using the spread syntax.

For example, the `push` method of arrays takes any number of arguments:

```
tsconst arr1 = [1, 2, 3];const arr2 = [4, 5, 6];arr1.push(...arr2);Try
```

Note that in general, TypeScript does not assume that arrays are immutable.

This can lead to some surprising behavior:

```
ts// Inferred type is number[] -- "an array with zero or more numbers",// not specifically two numbersconst args = [8, 5];const angle = Math.atan2(...args);A spread argument must either have a tuple type or be passed to a rest parameter.2556A spread argument must either have a tuple type or be passed to a rest parameter.Try
```

The best fix for this situation depends a bit on your code, but in general a `const` context is the most straightforward solution:

```
ts// Inferred as 2-length tupleconst args = [8, 5] as const;// OKconst angle = Math.atan2(...args);Try
```

Using rest arguments may require turning on `downlevelIteration` when targeting older runtimes.

## Parameter Destructuring

## Background Reading: Destructuring Assignment

You can use parameter destructuring to conveniently unpack objects provided as an argument into one or more local variables in the function body.

In JavaScript, it looks like this:

```
jsfunction sum({ a, b, c }) { console.log(a + b + c);}sum({ a: 10, b: 3, c: 9 });
```

The type annotation for the object goes after the destructuring syntax:

```
tsfunction sum({ a, b, c }: { a: number; b: number; c: number }) { console.log(a + b + c);}
```

Try

This can look a bit verbose, but you can use a named type here as well:

```
ts// Same as prior exampletype ABC = { a: number; b: number; c: number };function sum({ a, b, c }: ABC) { console.log(a + b + c);}
```

Assignability of Functions

Return type void

The void return type for functions can produce some unusual, but expected behavior.

Contextual typing with a return type of void does not force functions to not return something. Another way to say this is a contextual function type with a void return type (type voidFunc = () => void), when implemented, can return any other value, but it will be ignored.

Thus, the following implementations of the type () => void are valid:

```
ts// type voidFunc = () => void; const f1: voidFunc = () => { return true;}; const f2:
```

```
voidFunc = () => true; const f3: voidFunc = function () { return true;};
```

Try  
And when the return value of one of these functions is assigned to another variable, it will retain the type of void:

```
tsconst v1 = f1(); const v2 = f2(); const v3 = f3();
```

Try  
This behavior exists so that the following code is valid even though

Array.prototype.push returns a number and the Array.prototype.forEach method expects a function with a return type of void.

```
tsconst src = [1, 2, 3]; const dst = [0]; src.forEach((el) => dst.push(el));
```

Try  
There is one other special case to be aware of, when a literal function definition has a void return type, that function must not return anything.

```
tsfunction f2(): void { // @ts-expect-error return true;} const f3 = function (): void { //  
@ts-expect-error return true;};
```

Try  
For more on void please refer to these other documentation entries:

v1 handbook

v2 handbook

FAQ - "Why are functions returning non-void assignable to function returning void?"

In JavaScript, the fundamental way that we group and pass around data is through objects.

In TypeScript, we represent those through object types.

As we've seen, they can be anonymous:

```
tsfunction greet(person: { name: string; age: number }) { return "Hello " + person.name;}
```

Try

or they can be named by using either an interface:

```
tsinterface Person { name: string; age: number;} function greet(person: Person)
{ return "Hello " + person.name;}Try
```

or a type alias:

```
tstype Person = { name: string; age: number;}; function greet(person: Person) { return
"Hello " + person.name;}Try
```

In all three examples above, we've written functions that take objects that contain the property name (which must be a string) and age (which must be a number).

### Quick Reference

We have cheat-sheets available for both type and interface, if you want a quick look at the important every-day syntax at a glance.

### Property Modifiers

Each property in an object type can specify a couple of things: the type, whether the property is optional, and whether the property can be written to.

### Optional Properties

Much of the time, we'll find ourselves dealing with objects that might have a property set.

In those cases, we can mark those properties as optional by adding a question mark (?) to the end of their names.

```
tsinterface PaintOptions { shape: Shape; xPos?: number; yPos?: number;} function
paintShape(opts: PaintOptions) { // ...} const shape =
getShape();paintShape({ shape });paintShape({ shape, xPos:
100 });paintShape({ shape, yPos: 100 });paintShape({ shape, xPos: 100, yPos: 100 });Try
In this example, both xPos and yPos are considered optional.
```

We can choose to provide either of them, so every call above to paintShape is valid.

All optionality really says is that if the property is set, it better have a specific type.

We can also read from those properties - but when we do under strictNullChecks, TypeScript will tell us they're potentially undefined.

```
tsfunction paintShape(opts: PaintOptions) { let xPos = opts.xPos;                (property)
PaintOptions.xPos?: number | undefined let yPos = opts.yPos;                (property)
PaintOptions.yPos?: number | undefined // ...}Try
```

In JavaScript, even if the property has never been set, we can still access it - it's just going to give us the value undefined.

We can just handle undefined specially by checking for it.

```
tsfunction paintShape(opts: PaintOptions) { let xPos = opts.xPos === undefined ? 0 :
opts.xPos;    let xPos: number let yPos = opts.yPos === undefined ? 0 :
opts.yPos;    let yPos: number // ...}Try
```

Note that this pattern of setting defaults for unspecified values is so common that JavaScript has syntax to support it.

```
tsfunction paintShape({ shape, xPos = 0, yPos = 0 }: PaintOptions) { console.log("x
coordinate at", xPos);                (parameter) xPos: number console.log("y
coordinate at", yPos);                (parameter) yPos: number // ...}Try
```

Here we used a destructuring pattern for paintShape's parameter, and provided default values for xPos and yPos.

Now xPos and yPos are both definitely present within the body of paintShape, but optional for any callers to paintShape.

Note that there is currently no way to place type annotations within destructuring patterns.

This is because the following syntax already means something different in JavaScript.

```
tsfunction draw({ shape: Shape, xPos: number = 100 /*...*/ }) { render(shape);Cannot find name 'shape'. Did you mean 'Shape'?2552Cannot find name 'shape'. Did you mean 'Shape'? render(xPos);Cannot find name 'xPos'.2304Cannot find name 'xPos'.}Try
```

In an object destructuring pattern, `shape: Shape` means “grab the property `shape` and redefine it locally as a variable named `Shape`.”

Likewise `xPos: number` creates a variable named `number` whose value is based on the parameter’s `xPos`.

Using mapping modifiers, you can remove optional attributes.

### readonly Properties

Properties can also be marked as `readonly` for TypeScript.

While it won’t change any behavior at runtime, a property marked as `readonly` can’t be written to during type-checking.

```
tsinterface SomeType { readonly prop: string;} function doSomething(obj: SomeType) { // We can read from 'obj.prop'. console.log(`prop has the value '${obj.prop}'.`); // But we can't re-assign it. obj.prop = "hello";Cannot assign to 'prop' because it is a read-only property.2540Cannot assign to 'prop' because it is a read-only property.}Try
```

Using the `readonly` modifier doesn’t necessarily imply that a value is totally immutable - or in other words, that its internal contents can’t be changed.

It just means the property itself can’t be re-written to.

```
tsinterface Home { readonly resident: { name: string; age: number };} function visitForBirthday(home: Home) { // We can read and update properties from 'home.resident'. console.log(`Happy birthday ${home.resident.name}!`); home.resident.age++;} function evict(home: Home) { // But we can't write to the 'resident' property itself on a 'Home'. home.resident = {Cannot assign to 'resident' because it is a read-only property.2540Cannot assign to 'resident' because it is a read-only property. name: "Victor the Evictor", age: 42, };}Try
```

It’s important to manage expectations of what `readonly` implies.

It’s useful to signal intent during development time for TypeScript on how an object should be used.

TypeScript doesn’t factor in whether properties on two types are `readonly` when checking whether those types are compatible, so `readonly` properties can also change via aliasing.

```
tsinterface Person { name: string; age: number;} interface ReadonlyPerson { readonly name: string; readonly age: number;} let writablePerson: Person = { name: "Person McPersonface", age: 42,}; // workslet readonlyPerson: ReadonlyPerson = writablePerson; console.log(readonlyPerson.age); // prints '42'writablePerson.age++;console.log(readonlyPerson.age); // prints '43'Try
```

Using mapping modifiers, you can remove `readonly` attributes.

### Index Signatures

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

In those cases you can use an index signature to describe the types of possible values, for example:

```
tsinterface StringArray { [index: number]: string;} const myArray: StringArray =
getStringArray();const secondItem = myArray[1];      const secondItem: stringTry
```

Above, we have a StringArray interface which has an index signature.

This index signature states that when a StringArray is indexed with a number, it will return a string.

Only some types are allowed for index signature properties: string, number, symbol, template string patterns, and union types consisting only of these.

It is possible to support both types of indexers...

It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a number, JavaScript will actually convert that to a string before indexing into an object. That means that indexing with 100 (a number) is the same thing as indexing with "100" (a string), so the two need to be consistent.

```
tsinterface Animal { name: string;} interface Dog extends Animal { breed: string;} //
Error: indexing with a numeric string might get you a completely separate type of
Animal!interface NotOkay { [x: number]: Animal;'number' index type 'Animal' is not
assignable to 'string' index type 'Dog'.2413'number' index type 'Animal' is not
assignable to 'string' index type 'Dog'. [x: string]: Dog;}Try
```

While string index signatures are a powerful way to describe the “dictionary” pattern, they also enforce that all properties match their return type.

This is because a string index declares that obj.property is also available as obj["property"].

In the following example, name’s type does not match the string index’s type, and the type checker gives an error:

```
tsinterface NumberDictionary { [index: string]: number; length: number; // ok name:
string;Property 'name' of type 'string' is not assignable to 'string' index type
'number'.2411Property 'name' of type 'string' is not assignable to 'string' index type
'number'.}Try
```

However, properties of different types are acceptable if the index signature is a union of the property types:

```
tsinterface NumberOrStringDictionary { [index: string]: number | string; length:
number; // ok, length is a number name: string; // ok, name is a string}Try
```

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

```
tsinterface ReadonlyStringArray { readonly [index: number]: string;} let myArray:
ReadonlyStringArray = getReadOnlyStringArray();myArray[2] = "Mallory";Index
signature in type 'ReadonlyStringArray' only permits reading.2542Index signature in
type 'ReadonlyStringArray' only permits reading.Try
```

You can’t set myArray[2] because the index signature is readonly.

### Excess Property Checks

Where and how an object is assigned a type can make a difference in the type system. One of the key examples of this is in excess property checking, which validates the



object more thoroughly when it is created and assigned to an object type during creation.

```
tsinterface SquareConfig { color?: string; width?: number;} function  
createSquare(config: SquareConfig): { color: string; area: number } { return { color:  
config.color || "red", area: config.width ? config.width * config.width : 20, };} let  
mySquare = createSquare({ colour: "red", width: 100 });Argument of type '{ colour:  
string; width: number; }' is not assignable to parameter of type 'SquareConfig'.
```

Object literal may only specify known properties, but 'colour' does not exist in type 'SquareConfig'. Did you mean to write 'color'?2345Argument of type '{ colour: string; width: number; }' is not assignable to parameter of type 'SquareConfig'.

Object literal may only specify known properties, but 'colour' does not exist in type 'SquareConfig'. Did you mean to write 'color'?Try

Notice the given argument to createSquare is spelled colour instead of color.

In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the width properties are compatible, there's no color property present, and the extra colour property is insignificant.

However, TypeScript takes the stance that there's probably a bug in this code.

Object literals get special treatment and undergo excess property checking when assigning them to other variables, or passing them as arguments.

If an object literal has any properties that the "target type" doesn't have, you'll get an error:

```
tslet mySquare = createSquare({ colour: "red", width: 100 });Argument of type '{ colour:  
string; width: number; }' is not assignable to parameter of type 'SquareConfig'.
```

Object literal may only specify known properties, but 'colour' does not exist in type 'SquareConfig'. Did you mean to write 'color'?2345Argument of type '{ colour: string; width: number; }' is not assignable to parameter of type 'SquareConfig'.

Object literal may only specify known properties, but 'colour' does not exist in type 'SquareConfig'. Did you mean to write 'color'?Try

Getting around these checks is actually really simple.

The easiest method is to just use a type assertion:

```
tslet mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);Try
```

However, a better approach might be to add a string index signature if you're sure that the object can have some extra properties that are used in some special way.

If SquareConfig can have color and width properties with the above types, but could also have any number of other properties, then we could define it like so:

```
tsinterface SquareConfig { color?: string; width?: number; [propName: string]: any;}Try
```

We'll discuss index signatures in a bit, but here we're saying a SquareConfig can have any number of properties, and as long as they aren't color or width, their types don't matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable:

Since assigning squareOptions won't undergo excess property checks, the compiler won't give you an error:

```
tslet squareOptions = { colour: "red", width: 100 };let mySquare =  
createSquare(squareOptions);Try
```

The above workaround will work as long as you have a common property between `squareOptions` and `SquareConfig`.

In this example, it was the property `width`. It will however, fail if the variable does not have any common object property. For example:

```
tslet squareOptions = { colour: "red" };let mySquare =
createSquare(squareOptions);Type '{ colour: string; }' has no properties in common with
type 'SquareConfig'.2559Type '{ colour: string; }' has no properties in common with type
'SquareConfig'.Try
```

Keep in mind that for simple code like above, you probably shouldn't be trying to "get around" these checks.

For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs.

That means if you're running into excess property checking problems for something like option bags, you might need to revise some of your type declarations.

In this instance, if it's okay to pass an object with both a `color` or `colour` property to `createSquare`, you should fix up the definition of `SquareConfig` to reflect that.

### Extending Types

It's pretty common to have types that might be more specific versions of other types.

For example, we might have a `BasicAddress` type that describes the fields necessary for sending letters and packages in the U.S.

```
tsinterface BasicAddress { name?: string; street: string; city: string; country: string;
postalCode: string;}Try
```

In some situations that's enough, but addresses often have a unit number associated with them if the building at an address has multiple units.

We can then describe an `AddressWithUnit`.

```
tsinterface AddressWithUnit { name?: string; unit: string; street: string; city: string;
country: string; postalCode: string;}Try
```

This does the job, but the downside here is that we had to repeat all the other fields from `BasicAddress` when our changes were purely additive.

Instead, we can extend the original `BasicAddress` type and just add the new fields that are unique to `AddressWithUnit`.

```
tsinterface BasicAddress { name?: string; street: string; city: string; country: string;
postalCode: string;} interface AddressWithUnit extends BasicAddress { unit: string;}Try
```

The `extends` keyword on an interface allows us to effectively copy members from other named types, and add whatever new members we want.

This can be useful for cutting down the amount of type declaration boilerplate we have to write, and for signaling intent that several different declarations of the same property might be related.

For example, `AddressWithUnit` didn't need to repeat the `street` property, and because `street` originates from `BasicAddress`, a reader will know that those two types are related in some way.

Interfaces can also extend from multiple types.

```
tsinterface Colorful { color: string;} interface Circle { radius: number;} interface
ColorfulCircle extends Colorful, Circle {} const cc: ColorfulCircle = { color: "red", radius:
42,;}Try
```

## Intersection Types

interfaces allowed us to build up new types from other types by extending them.

TypeScript provides another construct called intersection types that is mainly used to combine existing object types.

An intersection type is defined using the & operator.

```
tsinterface Colorful { color: string;}interface Circle { radius: number;} type  
ColorfulCircle = Colorful & Circle;Try
```

Here, we've intersected Colorful and Circle to produce a new type that has all the members of Colorful and Circle.

```
tsfunction draw(circle: Colorful & Circle) { console.log(`Color was ${circle.color}`);  
console.log(`Radius was ${circle.radius}`);} // okaydraw({ color: "blue", radius: 42 }); //  
oopsdraw({ color: "red", radius: 42 });Argument of type '{ color: string; radius: number; }'  
is not assignable to parameter of type 'Colorful & Circle'.
```

Object literal may only specify known properties, but 'radius' does not exist in type 'Colorful & Circle'. Did you mean to write 'radius'?2345Argument of type '{ color: string; radius: number; }' is not assignable to parameter of type 'Colorful & Circle'.

Object literal may only specify known properties, but 'radius' does not exist in type 'Colorful & Circle'. Did you mean to write 'radius'?Try

## Interfaces vs. Intersections

We just looked at two ways to combine types which are similar, but are actually subtly different.

With interfaces, we could use an extends clause to extend from other types, and we were able to do something similar with intersections and name the result with a type alias.

The principal difference between the two is how conflicts are handled, and that difference is typically one of the main reasons why you'd pick one over the other between an interface and a type alias of an intersection type.

## Generic Object Types

Let's imagine a Box type that can contain any value - strings, numbers, Giraffes, whatever.

```
tsinterface Box { contents: any;}Try
```

Right now, the contents property is typed as any, which works, but can lead to accidents down the line.

We could instead use unknown, but that would mean that in cases where we already know the type of contents, we'd need to do precautionary checks, or use error-prone type assertions.

```
tsinterface Box { contents: unknown;} let x: Box = { contents: "hello world",}; // we could  
check 'x.contents' if (typeof x.contents === "string")  
{ console.log(x.contents.toLowerCase());} // or we could use a type  
assertionconsole.log((x.contents as string).toLowerCase());Try
```

One type safe approach would be to instead scaffold out different Box types for every type of contents.

```
tsinterface NumberBox { contents: number;} interface StringBox { contents:  
string;} interface BooleanBox { contents: boolean;}Try
```

But that means we'll have to create different functions, or overloads of functions, to

operate on these types.

```
tsfunction setContents(box: StringBox, newContents: string): void;function  
setContents(box: NumberBox, newContents: number): void;function setContents(box:  
BooleanBox, newContents: boolean): void;function setContents(box: { contents: any },  
newContents: any) { box.contents = newContents;}Try
```

That's a lot of boilerplate. Moreover, we might later need to introduce new types and overloads.

This is frustrating, since our box types and overloads are all effectively the same.

Instead, we can make a generic Box type which declares a type parameter.

```
tsinterface Box<Type> { contents: Type;}Try
```

You might read this as “A Box of Type is something whose contents have type Type”.

Later on, when we refer to Box, we have to give a type argument in place of Type.

```
tslet box: Box<string>;Try
```

Think of Box as a template for a real type, where Type is a placeholder that will get replaced with some other type.

When TypeScript sees Box<string>, it will replace every instance of Type in Box<Type> with string, and end up working with something like { contents: string }.

In other words, Box<string> and our earlier StringBox work identically.

```
tsinterface Box<Type> { contents: Type;}interface StringBox { contents: string;} let  
boxA: Box<string> = { contents: "hello" };boxA.contents;      (property)
```

```
Box<string>.contents: string let boxB: StringBox = { contents:  
"world" };boxB.contents;      (property) StringBox.contents: stringTry
```

Box is reusable in that Type can be substituted with anything. That means that when we need a box for a new type, we don't need to declare a new Box type at all (though we certainly could if we wanted to).

```
tsinterface Box<Type> { contents: Type;} interface Apple { // ....} // Same as '{ contents:  
Apple }'.type AppleBox = Box<Apple>;Try
```

This also means that we can avoid overloads entirely by instead using generic functions.

```
tsfunction setContents<Type>(box: Box<Type>, newContents: Type) { box.contents =  
newContents;}Try
```

It is worth noting that type aliases can also be generic. We could have defined our new Box<Type> interface, which was:

```
tsinterface Box<Type> { contents: Type;}Try
```

by using a type alias instead:

```
tsinterface Box<Type> = { contents: Type;}Try
```

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

```
tsinterface OrNull<Type> = Type | null; type OneOrMany<Type> = Type | Type[]; type
```

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

```
OneOrManyOrNull<Type> = OneOrMany<Type> | null type OneOrManyOrNullStrings =  
OneOrManyOrNull<string>;      type OneOrManyOrNullStrings =
```

```
OneOrMany<string> | nullTry
```

We'll circle back to type aliases in just a little bit.

The Array Type

Generic object types are often some sort of container type that work independently of the type of elements they contain.

It's ideal for data structures to work this way so that they're re-usable across different data types.

It turns out we've been working with a type just like that throughout this handbook: the Array type.

Whenever we write out types like `number[]` or `string[]`, that's really just a shorthand for `Array<number>` and `Array<string>`.

```
tsfunction doSomething(value: Array<string>) { // ...} let myArray: string[] = ["hello", "world"]; // either of these work!doSomething(myArray);doSomething(new Array("hello", "world"));Try
```

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

```
tsinterface Array<Type> { /** * Gets or sets the length of the array. */ length: number; /** * Removes the last element from an array and returns it. */ pop(): Type | undefined; /** * Appends new elements to an array, and returns the new length of the array. */ push(...items: Type[]): number; // ...}Try
```

Modern JavaScript also provides other data structures which are generic, like `Map<K, V>`, `Set<T>`, and `Promise<T>`.

All this really means is that because of how Map, Set, and Promise behave, they can work with any sets of types.

### The ReadonlyArray Type

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

```
tsfunction doStuff(values: ReadonlyArray<string>) { // We can read from 'values'... const copy = values.slice(); console.log(`The first value is ${values[0]}`); // ...but we can't mutate 'values'. values.push("hello!");Property 'push' does not exist on type 'readonly string[]'.2339Property 'push' does not exist on type 'readonly string[]'.}Try
```

Much like the readonly modifier for properties, it's mainly a tool we can use for intent.

When we see a function that returns ReadonlyArrays, it tells us we're not meant to change the contents at all, and when we see a function that consumes

ReadonlyArrays, it tells us that we can pass any array into that function without worrying that it will change its contents.

Unlike Array, there isn't a ReadonlyArray constructor that we can use.

```
tsnew ReadonlyArray("red", "green", "blue");ReadonlyArray' only refers to a type, but is being used as a value here.2693'ReadonlyArray' only refers to a type, but is being used as a value here.Try
```

Instead, we can assign regular Arrays to ReadonlyArrays.

```
tsconst roArray: ReadonlyArray<string> = ["red", "green", "blue"];Try
```

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

```
tsfunction doStuff(values: readonly string[]) { // We can read from 'values'... const copy = values.slice(); console.log(`The first value is ${values[0]}`); // ...but we can't mutate 'values'. values.push("hello!");Property 'push' does not exist on type 'readonly string[]'.2339Property 'push' does not exist on type 'readonly string[]'.}Try
```

One last thing to note is that unlike the readonly property modifier, assignability isn't bidirectional between regular Arrays and ReadonlyArrays.

```
tslet x: readonly string[] = [];let y: string[] = []; x = y;y = x;The type 'readonly string[]' is 'readonly' and cannot be assigned to the mutable type 'string[]'.4104The type 'readonly string[]' is 'readonly' and cannot be assigned to the mutable type 'string[]'.Try
```

## Tuple Types

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

```
tstype StringNumberPair = [string, number];Try
```

Here, StringNumberPair is a tuple type of string and number.

Like ReadonlyArray, it has no representation at runtime, but is significant to TypeScript.

To the type system, StringNumberPair describes arrays whose 0 index contains a string and whose 1 index contains a number.

```
tsfunction doSomething(pair: [string, number]) { const a = pair[0];    const a: string
const b = pair[1];    const b: number // ...} doSomething(["hello", 42]);Try
```

If we try to index past the number of elements, we'll get an error.

```
tsfunction doSomething(pair: [string, number]) { // ... const c = pair[2];Tuple type
'[string, number]' of length '2' has no element at index '2'.2493Tuple type '[string,
number]' of length '2' has no element at index '2'.}Try
```

We can also destructure tuples using JavaScript's array destructuring.

```
tsfunction doSomething(stringHash: [string, number]) { const [inputString, hash] =
stringHash; console.log(inputString);    const inputString: string
console.log(hash);    const hash: number}Try
```

Tuple types are useful in heavily convention-based APIs, where each element's meaning is "obvious".

This gives us flexibility in whatever we want to name our variables when we destructure them.

In the above example, we were able to name elements 0 and 1 to whatever we wanted. However, since not every user holds the same view of what's obvious, it may be worth reconsidering whether using objects with descriptive property names may be better for your API.

Other than those length checks, simple tuple types like these are equivalent to types which are versions of Arrays that declare properties for specific indexes, and that declare length with a numeric literal type.

```
tsinterface StringNumberPair { // specialized properties length: 2; 0: string; 1:
number; // Other 'Array<string | number>' members... slice(start?: number, end?:
number): Array<string | number>;}Try
```

Another thing you may be interested in is that tuples can have optional properties by writing out a question mark (? after an element's type).

Optional tuple elements can only come at the end, and also affect the type of length.

```
tstype Either2dOr3d = [number, number, number?]; function setCoordinate(coord:
Either2dOr3d) { const [x, y, z] = coord;    const z: number | undefined
console.log(` Provided coordinates had ${coord.length}
dimensions`);    (property) length: 2 | 3}Try
```

Tuples can also have rest elements, which have to be an array/tuple type.

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

```
StringBooleansNumber = [string, ...boolean[], number];type BooleansStringNumber =
[...boolean[], string, number];Try
```

StringNumberBooleans describes a tuple whose first two elements are string and number respectively, but which may have any number of booleans following. StringBooleansNumber describes a tuple whose first element is string and then any number of booleans and ending with a number. BooleansStringNumber describes a tuple whose starting elements are any number of booleans and ending with a string then a number.

A tuple with a rest element has no set “length” - it only has a set of well-known elements in different positions.

```
tsconst a: StringNumberBooleans = ["hello", 1];const b: StringNumberBooleans = ["beautiful", 2, true];const c: StringNumberBooleans = ["world", 3, true, false, true, false, true];Try
```

Why might optional and rest elements be useful?

Well, it allows TypeScript to correspond tuples with parameter lists.

Tuples types can be used in rest parameters and arguments, so that the following:

```
tsfunction readButtonInput(...args: [string, number, ...boolean[]]) { const [name, version, ...input] = args; // ...}Try
```

is basically equivalent to:

```
tsfunction readButtonInput(name: string, version: number, ...input: boolean[]) { // ...}Try
```

This is handy when you want to take a variable number of arguments with a rest parameter, and you need a minimum number of elements, but you don't want to introduce intermediate variables.

## readonly Tuple Types

One final note about tuple types - tuple types have readonly variants, and can be specified by sticking a readonly modifier in front of them - just like with array shorthand syntax.

```
tsfunction doSomething(pair: readonly [string, number]) { // ...}Try
```

As you might expect, writing to any property of a readonly tuple isn't allowed in TypeScript.

```
tsfunction doSomething(pair: readonly [string, number]) { pair[0] = "hello!";Cannot assign to '0' because it is a read-only property.2540Cannot assign to '0' because it is a read-only property.}Try
```

Tuples tend to be created and left un-modified in most code, so annotating types as readonly tuples when possible is a good default.

This is also important given that array literals with const assertions will be inferred with readonly tuple types.

```
tslet point = [3, 4] as const; function distanceFromOrigin([x, y]: [number, number]) { return Math.sqrt(x ** 2 + y ** 2);} distanceFromOrigin(point);Argument of type 'readonly [3, 4]' is not assignable to parameter of type '[number, number]'.
```

The type 'readonly [3, 4]' is 'readonly' and cannot be assigned to the mutable type '[number, number]'.2345Argument of type 'readonly [3, 4]' is not assignable to parameter of type '[number, number]'.

The type 'readonly [3, 4]' is 'readonly' and cannot be assigned to the mutable type '[number, number]'.Try

Here, distanceFromOrigin never modifies its elements, but expects a mutable tuple.

Since point's type was inferred as readonly [3, 4], it won't be compatible with [number, number] since that type can't guarantee point's elements won't be mutated.

TypeScript's type system is very powerful because it allows expressing types in terms of other types.

The simplest form of this idea is generics. Additionally, we have a wide variety of type operators available to use.

It's also possible to express types in terms of values that we already have.

By combining various type operators, we can express complex operations and values in a succinct, maintainable way.

In this section we'll cover ways to express a new type in terms of an existing type or value.

Generics - Types which take parameters

Keyof Type Operator - Using the keyof operator to create new types

typeof Type Operator - Using the typeof operator to create new types

Indexed Access Types - Using Type['a'] syntax to access a subset of a type

Conditional Types - Types which act like if statements in the type system

Mapped Types - Creating types by mapping each property in an existing type

Template Literal Types - Mapped types which change properties via template literal strings

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable.

Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able to create a component that can work over a variety of types rather than a single one.

This allows users to consume these components and use their own types.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function.

The identity function is a function that will return back whatever is passed in.

You can think of this in a similar way to the echo command.

Without generics, we would either have to give the identity function a specific type:

```
tsfunction identity(arg: number): number { return arg;}Try
```

Or, we could describe the identity function using the any type:

```
tsfunction identity(arg: any): any { return arg;}Try
```

While using any is certainly generic in that it will cause the function to accept any and all types for the type of arg, we actually are losing the information about what that type was when the function returns.

If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned.

Here, we will use a type variable, a special kind of variable that works on types rather than values.



```
tsfunction identity<Type>(arg: Type): Type { return arg;}Try
```

We've now added a type variable `Type` to the identity function.

This `Type` allows us to capture the type the user provides (e.g. number), so that we can use that information later.

Here, we use `Type` again as the return type. On inspection, we can now see the same type is used for the argument and the return type.

This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the identity function is generic, as it works over a range of types.

Unlike using any, it's also just as precise (i.e., it doesn't lose any information) as the first identity function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways.

The first way is to pass all of the arguments, including the type argument, to the function:

```
tslet output = identity<string>("myString");    let output: stringTry
```

Here we explicitly set `Type` to be `string` as one of the arguments to the function call, denoted using the `<>` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use type argument inference — that is, we want the compiler to set the value of `Type` for us automatically based on the type of the argument we pass in:

```
tslet output = identity("myString");    let output: stringTry
```

Notice that we didn't have to explicitly pass the type in the angle brackets (`<>`); the compiler just looked at the value `"myString"`, and set `Type` to its type.

While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

### Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly.

That is, that you actually treat these parameters as if they could be any and all types.

Let's take our identity function from earlier:

```
tsfunction identity<Type>(arg: Type): Type { return arg;}Try
```

What if we want to also log the length of the argument `arg` to the console with each call?

We might be tempted to write this:

```
tsfunction loggingIdentity<Type>(arg: Type): Type { console.log(arg.length);Property  
'length' does not exist on type 'Type'.2339Property 'length' does not exist on type  
'Type'. return arg;}Try
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member.

Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a number instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `Type` rather than

Type directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

```
tsfunction loggingIdentity<Type>(arg: Type[]): Type[] { console.log(arg.length); return arg;}Try
```

You can read the type of `loggingIdentity` as “the generic function `loggingIdentity` takes a type parameter `Type`, and an argument `arg` which is an array of `Types`, and returns an array of `Types`.”

If we passed in an array of numbers, we'd get an array of numbers back out, as `Type` would bind to `number`.

This allows us to use our generic type variable `Type` as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
tsfunction loggingIdentity<Type>(arg: Array<Type>): Array<Type>
{ console.log(arg.length); // Array has a .length, so no more error return arg;}Try
```

You may already be familiar with this style of type from other languages.

In the next section, we'll cover how you can create your own generic types like `Array<Type>`.

## Generic Types

In previous sections, we created generic identity functions that worked over a range of types.

In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

```
tsfunction identity<Type>(arg: Type): Type { return arg;} let myIdentity: <Type>(arg:
Type) => Type = identity;Try
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

```
tsfunction identity<Input>(arg: Input): Input { return arg;} let myIdentity: <Input>(arg:
Input) => Input = identity;Try
```

We can also write the generic type as a call signature of an object literal type:

```
tsfunction identity<Type>(arg: Type): Type { return arg;} let myIdentity: { <Type>(arg:
Type): Type } = identity;Try
```

Which leads us to writing our first generic interface.

Let's take the object literal from the previous example and move it to an interface:

```
tsinterface GenericIdentityFn { <Type>(arg: Type): Type;} function identity<Type>(arg:
Type): Type { return arg;} let myIdentity: GenericIdentityFn = identity;Try
```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface.

This lets us see what type(s) we're generic over (e.g. `Dictionary<string>` rather than just `Dictionary`).

This makes the type parameter visible to all the other members of the interface.

```
tsinterface GenericIdentityFn<Type> { (arg: Type): Type;} function identity<Type>(arg:
Type): Type { return arg;} let myIdentity: GenericIdentityFn<number> = identity;Try
```

Notice that our example has changed to be something slightly different.

Instead of describing a generic function, we now have a non-generic function signature

that is a part of a generic type.

When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use.

Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes.

Note that it is not possible to create generic enums and namespaces.

### Generic Classes

A generic class has a similar shape to a generic interface.

Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```
tsclass GenericNumber<NumType> { zeroValue: NumType; add: (x: NumType, y: NumType) => NumType; } let myGenericNumber = new GenericNumber<number>(); myGenericNumber.zeroValue = 0; myGenericNumber.add = function (x, y) { return x + y; }; Try
```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type.

We could have instead used `string` or even more complex objects.

```
tslet stringNumeric = new GenericNumber<string>(); stringNumeric.zeroValue = ""; stringNumeric.add = function (x, y) { return x + y; }; console.log(stringNumeric.add(stringNumeric.zeroValue, "test")); Try
```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we cover in our section on classes, a class has two sides to its type: the static side and the instance side.

Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

### Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have.

In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

```
tsfunction loggingIdentity<Type>(arg: Type): Type { console.log(arg.length); Property 'length' does not exist on type 'Type'. } return arg; } Try
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that also have the `.length` property.

As long as the type has this member, we'll allow it, but it's required to have at least this member.

To do so, we must list our requirement as a constraint on what `Type` can be.

To do so, we'll create an interface that describes our constraint.

Here, we'll create an interface that has a single `.length` property and then we'll use this

interface and the extends keyword to denote our constraint:

```
tsinterface Lengthwise { length: number;} function loggingIdentity<Type extends Lengthwise>(arg: Type): Type { console.log(arg.length); // Now we know it has a .length property, so no more error return arg;}Try
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
tsloggingIdentity(3);Argument of type 'number' is not assignable to parameter of type 'Lengthwise'.2345Argument of type 'number' is not assignable to parameter of type 'Lengthwise'.Try
```

Instead, we need to pass in values whose type has all the required properties:

```
tsloggingIdentity({ length: 10, value: 3 });Try
```

### Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter.

For example, here we'd like to get a property from an object given its name.

We'd like to ensure that we're not accidentally grabbing a property that does not exist on the obj, so we'll place a constraint between the two types:

```
tsfunction getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) { return obj[key];} let x = { a: 1, b: 2, c: 3, d: 4 }; getProperty(x, "a");getProperty(x, "m");Argument of type '"m"' is not assignable to parameter of type '"a" | "b" | "c" | "d"'.2345Argument of type '"m"' is not assignable to parameter of type '"a" | "b" | "c" | "d"'.Try
```

### Using Class Types in Generics

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

```
tsfunction create<Type>(c: { new (): Type }): Type { return new c();}Try
```

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

```
tsclass BeeKeeper { hasMask: boolean = true;} class ZooKeeper { nametag: string = "Mikle";} class Animal { numLegs: number = 4;} class Bee extends Animal { numLegs = 6; keeper: BeeKeeper = new BeeKeeper();} class Lion extends Animal { keeper: ZooKeeper = new ZooKeeper();} function createInstance<A extends Animal>(c: new () => A): A { return new
```

```
c();} createInstance(Lion).keeper.nametag;createInstance(Bee).keeper.hasMask;Try
```

This pattern is used to power the mixins design pattern.

### Generic Parameter Defaults

Consider a function that creates a new HTMLElement. Calling the function with no arguments generates a Div; calling it with an element as the first argument generates an element of the argument's type. You can optionally pass a list of children as well.

Previously you would have to define it as:

```
tsdeclare function create(): Container<HTMLDivElement, HTMLDivElement[]>;Cannot find name 'Container'.2304Cannot find name 'Container'.declare function create<T extends HTMLElement>(element: T): Container<T, T[]>;Cannot find name 'Container'.2304Cannot find name 'Container'.declare function create<T extends HTMLElement, U extends HTMLElement>( element: T, children: U[]): Container<T, U[]>;Cannot find name 'Container'.2304Cannot find name 'Container'.Try
```

With generic parameter defaults we can reduce it to:

```
tsdeclare function create<T extends HTMLElement = HTMLDivElement, U =
```

```
T[]>( element?: T, children?: U): Container<T, U>;
```

Cannot find name 'Container'.2304

Cannot find name 'Container'.Try

A generic parameter default follows the following rules:

A type parameter is deemed optional if it has a default.

Required type parameters must not follow optional type parameters.

Default types for a type parameter must satisfy the constraint for the type parameter, if it exists.

When specifying type arguments, you are only required to specify type arguments for the required type parameters. Unspecified type parameters will resolve to their default types.

If a default type is specified and inference cannot choose a candidate, the default type is inferred.

A class or interface declaration that merges with an existing class or interface declaration may introduce a default for an existing type parameter.

A class or interface declaration that merges with an existing class or interface declaration may introduce a new type parameter as long as it specifies a default.

The keyof type operator

The keyof operator takes an object type and produces a string or numeric literal union of its keys.

The following type P is the same type as type P = "x" | "y":

```
type Point = { x: number; y: number };type P = keyof Point; type P = keyof Point
```

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

```
type Arrayish = { [n: number]: unknown };type A = keyof Arrayish; type A = number
type Mapish = { [k: string]: boolean };type M = keyof Mapish; type M = string | number
```

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

keyof types become especially useful when combined with mapped types, which we'll learn more about later.

The typeof type operator

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

```
// Prints "string"console.log(typeof "Hello world");
```

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

```
let s = "hello";let n: typeof s; let n: string
```

This isn't very useful for basic types, but combined with other type operators, you can use typeof to conveniently express many patterns.

For an example, let's start by looking at the predefined type ReturnType<T>.

It takes a function type and produces its return type:

```
type Predicate = (x: unknown) => boolean;type K = ReturnType<Predicate>; type K = boolean
```

If we try to use ReturnType on a function name, we see an instructive error:

```
function f() { return { x: 10, y: 3 };}type P = ReturnType<f>;
```

'f' refers to a value, but is being used as a type here. Did you mean 'typeof f'?2749

'f' refers to a value, but is being used as a type here. Did you mean 'typeof f'?Try

Remember that values and types aren't the same thing.

To refer to the type that the value `f` has, we use `typeof`:

```
tsfunction f() { return { x: 10, y: 3 };}type P = ReturnType<typeof f>; type P = {
  x: number;
  y: number;
}Try
```

### Limitations

TypeScript intentionally limits the sorts of expressions you can use `typeof` on.

Specifically, it's only legal to use `typeof` on identifiers (i.e. variable names) or their properties.

This helps avoid the confusing trap of writing code you think is executing, but isn't:

```
ts// Meant to use = ReturnType<typeof msgbox>let shouldContinue: typeof
msgbox("Are you sure you want to continue?");', ' expected.1005', ' expected.Try
```

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

```
tsinterface Person { age: number; name: string; alive: boolean };type Age =
Person["age"]; type Age = numberTry
```

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

```
tsinterface Person { age: number; name: string; alive: boolean };type I1 = Person["age" | "name"]; type I1 = string | number type I2 = Person[keyof
Person]; type I2 = string | number | boolean type AliveOrName = "alive" | "name";type
I3 = Person[AliveOrName]; type I3 = string | booleanTry
```

You'll even see an error if you try to index a property that doesn't exist:

```
tsinterface Person { age: number; name: string; alive: boolean };type I1 = Person["alve"];Property 'alve' does not exist on type 'Person'.2339Property
'alve' does not exist on type 'Person'.Try
```

Another example of indexing with an arbitrary type is using `number` to get the type of an array's elements.

We can combine this with `typeof` to conveniently capture the element type of an array literal:

```
tsconst MyArray = [ { name: "Alice", age: 15 }, { name: "Bob", age: 23 }, { name: "Eve",
age: 38 },]; type Person = typeof MyArray[number]; type Person = {
  name: string;
  age: number;
}type Age = typeof MyArray[number]["age"]; type Age = number// Or type Age2 =
Person["age"]; type Age2 = numberTry
```

You can only use types when indexing, meaning you can't use a `const` to make a variable reference:

```
tsconst key = "age";type Age = Person[key];Type 'key' cannot be used as an index
type.'key' refers to a value, but is being used as a type here. Did you mean 'typeof key'?
25382749Type 'key' cannot be used as an index type.'key' refers to a value, but is being
used as a type here. Did you mean 'typeof key'?Try
```

However, you can use a type alias for a similar style of refactor:

```
tsinterface Person { age: number; name: string; alive: boolean };type key = "age";type Age = Person[key];Try
```

At the heart of most useful programs, we have to make decisions based on input.

JavaScript programs are no different, but given the fact that values can be easily introspected, those decisions are also based on the types of the inputs.

Conditional types help describe the relation between the types of inputs and outputs.

```
tsinterface Animal { live(): void;}interface Dog extends Animal { woof(): void;} type
```

Example1 = Dog extends Animal ? number : string;      type Example1 = number type  
Example2 = RegExp extends Animal ? number : string;      type Example2 = stringTry  
Conditional types take a form that looks a little like conditional expressions (condition ? trueExpression : falseExpression) in JavaScript:

```
ts SomeType extends OtherType ? TrueType : FalseType;Try
```

When the type on the left of the extends is assignable to the one on the right, then you'll get the type in the first branch (the "true" branch); otherwise you'll get the type in the latter branch (the "false" branch).

From the examples above, conditional types might not immediately seem useful - we can tell ourselves whether or not Dog extends Animal and pick number or string!

But the power of conditional types comes from using them with generics.

For example, let's take the following createLabel function:

```
tsinterface IdLabel { id: number /* some fields */;}interface NameLabel { name: string /*  
other fields */;}function createLabel(id: number): IdLabel;function createLabel(name:  
string): NameLabel;function createLabel(nameOrId: string | number): IdLabel |  
NameLabel;function createLabel(nameOrId: string | number): IdLabel | NameLabel  
{ throw "unimplemented";}Try
```

These overloads for createLabel describe a single JavaScript function that makes a choice based on the types of its inputs. Note a few things:

If a library has to make the same sort of choice over and over throughout its API, this becomes cumbersome.

We have to create three overloads: one for each case when we're sure of the type (one for string and one for number), and one for the most general case (taking a string | number). For every new type createLabel can handle, the number of overloads grows exponentially.

Instead, we can encode that logic in a conditional type:

```
ts type NameOrId<T extends number | string> = T extends number ? IdLabel :  
NameLabel;Try
```

We can then use that conditional type to simplify our overloads down to a single function with no overloads.

```
tsfunction createLabel<T extends number | string>(idOrName: T): NameOrId<T>  
{ throw "unimplemented";}let a = createLabel("typescript"); let a: NameLabel let b =  
createLabel(2.8); let b: IdLabel let c = createLabel(Math.random() ? "hello" : 42);let c:  
NameLabel | IdLabelTry
```

### Conditional Type Constraints

Often, the checks in a conditional type will provide us with some new information.

Just like narrowing with type guards can give us a more specific type, the true branch of a conditional type will further constrain generics by the type we check against.

For example, let's take the following:

```
ts type MessageOf<T> = T["message"];Type '"message"' cannot be used to index type  
'T'.2536Type '"message"' cannot be used to index type 'T'.Try
```

In this example, TypeScript errors because T isn't known to have a property called message.

We could constrain T, and TypeScript would no longer complain:

```
tstype MessageOf<T extends { message: unknown }> = T["message"]; interface Email
{ message: string;} type EmailMessageContents = MessageOf<Email>;          type
EmailMessageContents = stringTry
```

However, what if we wanted MessageOf to take any type, and default to something like never if a message property isn't available?

We can do this by moving the constraint out and introducing a conditional type:

```
tstype MessageOf<T> = T extends { message: unknown } ? T["message"] :
never; interface Email { message: string;} interface Dog { bark(): void;} type
EmailMessageContents = MessageOf<Email>;          type EmailMessageContents =
string type DogMessageContents = MessageOf<Dog>;          type
DogMessageContents = neverTry
```

Within the true branch, TypeScript knows that T will have a message property.

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

```
tstype Flatten<T> = T extends any[] ? T[number] : T; // Extracts out the element
type.type Str = Flatten<string[]>;    type Str = string // Leaves the type alone.type Num
= Flatten<number>;    type Num = numberTry
```

When Flatten is given an array type, it uses an indexed access with number to fetch out string[]'s element type.

Otherwise, it just returns the type it was given.

### Inferring Within Conditional Types

We just found ourselves using conditional types to apply constraints and then extract out types.

This ends up being such a common operation that conditional types make it easier.

Conditional types provide us with a way to infer from types we compare against in the true branch using the infer keyword.

For example, we could have inferred the element type in Flatten instead of fetching it out “manually” with an indexed access type:

```
tstype Flatten<Type> = Type extends Array<infer Item> ? Item : Type;Try
```

Here, we used the infer keyword to declaratively introduce a new generic type variable named Item instead of specifying how to retrieve the element type of Type within the true branch.

This frees us from having to think about how to dig through and probing apart the structure of the types we're interested in.

We can write some useful helper type aliases using the infer keyword.

For example, for simple cases, we can extract the return type out from function types:

```
tstype GetReturnType<Type> = Type extends (...args: never[]) => infer Return ?
Return : never; type Num = GetReturnType<() => number>;    type Num =
number type Str = GetReturnType<(x: string) => string>;    type Str = string type Bools
= GetReturnType<(a: boolean, b: boolean) => boolean[]>;    type Bools = boolean[]Try
```

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

```
tsdeclare function stringOrNum(x: string): number;declare function stringOrNum(x:
number): string;declare function stringOrNum(x: string | number): string | number; type
```



```
T1 = ReturnType<typeof stringOrNum>; type T1 = string | numberTry
```

### Distributive Conditional Types

When conditional types act on a generic type, they become distributive when given a union type.

For example, take the following:

```
tstype ToArray<Type> = Type extends any ? Type[] : never;Try
```

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

```
tstype ToArray<Type> = Type extends any ? Type[] : never; type StrArrOrNumArr =  
ToArray<string | number>; type StrArrOrNumArr = string[] | number[]Try
```

What happens here is that ToArray distributes on:

```
ts string | number;Try
```

and maps over each member type of the union, to what is effectively:

```
ts ToArray<string> | ToArray<number>;Try
```

which leaves us with:

```
ts string[] | number[];Try
```

Typically, distributivity is the desired behavior.

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

```
tstype ToArrayNonDist<Type> = [Type] extends [any] ? Type[] : never; //
```

```
'StrArrOrNumArr' is no longer a union.type StrArrOrNumArr = ToArrayNonDist<string |  
number>; type StrArrOrNumArr = (string | number)[]Try
```

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

Mapped types build on the syntax for index signatures, which are used to declare the types of properties which have not been declared ahead of time:

```
tstype OnlyBoolsAndHorses = { [key: string]: boolean | Horse; }; const conforms:  
OnlyBoolsAndHorses = { del: true, rodney: false, };Try
```

A mapped type is a generic type which uses a union of PropertyKeys (frequently created via a keyof) to iterate through keys to create a type:

```
tstype OptionsFlags<Type> = { [Property in keyof Type]: boolean; };Try
```

In this example, OptionsFlags will take all the properties from the type Type and change their values to be a boolean.

```
tstype Features = { darkMode: () => void; newUserProfile: () => void; }; type  
FeatureOptions = OptionsFlags<Features>; type FeatureOptions = {  
  darkMode: boolean;  
  newUserProfile: boolean;  
}Try
```

### Mapping Modifiers

There are two additional modifiers which can be applied during mapping: readonly and ? which affect mutability and optionality respectively.

You can remove or add these modifiers by prefixing with - or +. If you don't add a prefix, then + is assumed.

```
ts// Removes 'readonly' attributes from a type's propertiestype CreateMutable<Type> =  
{ -readonly [Property in keyof Type]: Type[Property]; }; type LockedAccount = { readonly  
id: string; readonly name: string; }; type UnlockedAccount =
```

```

CreateMutable<LockedAccount>;      type UnlockedAccount = {
  id: string;
  name: string;
}Try
ts// Removes 'optional' attributes from a type's properties
type Concrete<Type> =
{ [Property in keyof Type]-?: Type[Property]; }; type MaybeUser = { id: string; name?:
string; age?: number; }; type User = Concrete<MaybeUser>;    type User = {
  id: string;
  name: string;
  age: number;
}Try

```

Key Remapping via as

In TypeScript 4.1 and onwards, you can re-map keys in mapped types with an as clause in a mapped type:

```

tstype MappedTypeWithNewProperties<Type> = { [Properties in keyof Type as
NewKeyType]: Type[Properties]}

```

You can leverage features like template literal types to create new property names from prior ones:

```

tstype Getters<Type> = { [Property in keyof Type as `get${Capitalize<string &
Property>}`]: () => Type[Property] }; interface Person { name: string; age: number;
location: string; } type LazyPerson = Getters<Person>;    type LazyPerson = {
  getName: () => string;
  getAge: () => number;
  getLocation: () => string;
}Try

```

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

```

ts// Remove the 'kind' property
type RemoveKindField<Type> = { [Property in keyof
Type as Exclude<Property, "kind">]: Type[Property] }; interface Circle { kind: "circle";
radius: number; } type KindlessCircle = RemoveKindField<Circle>;    type
KindlessCircle = {
  radius: number;
}Try

```

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

```

tstype EventConfig<Events extends { kind: string }> = { [E in Events as E["kind"]]:
(event: E) => void; } type SquareEvent = { kind: "square", x: number, y: number }; type
CircleEvent = { kind: "circle", radius: number }; type Config = EventConfig<SquareEvent
| CircleEvent>    type Config = {
  square: (event: SquareEvent) => void;
  circle: (event: CircleEvent) => void;
}Try

```

Further Exploration

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

```

tstype ExtractPII<Type> = { [Property in keyof Type]: Type[Property] extends { pii:

```

```

true } ? true : false;}; type DBFields = { id: { format: "incrementing" }; name: { type:
string; pii: true };}; type ObjectsNeedingGDPRDeletion =
ExtractPII<DBFields>;          type ObjectsNeedingGDPRDeletion = {
  id: false;
  name: true;
}Try

```

Template literal types build on string literal types, and have the ability to expand into many strings via unions.

They have the same syntax as template literal strings in JavaScript, but are used in type positions.

When used with concrete literal types, a template literal produces a new string literal type by concatenating the contents.

```

tstype World = "world"; type Greeting = `hello ${World}`;    type Greeting = "hello
world"Try

```

When a union is used in the interpolated position, the type is the set of every possible string literal that could be represented by each union member:

```

tstype EmailLocaleIDs = "welcome_email" | "email_heading";type FooterLocaleIDs =
"footer_title" | "footer_sendoff"; type AllLocaleIDs = `${EmailLocaleIDs |
FooterLocaleIDs}_id`;    type AllLocaleIDs = "welcome_email_id" |
"email_heading_id" | "footer_title_id" | "footer_sendoff_id"Try

```

For each interpolated position in the template literal, the unions are cross multiplied:

```

tstype AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;type Lang = "en" | "ja" |
"pt"; type LocaleMessageIDs = `${Lang}_${AllLocaleIDs}`;    type
LocaleMessageIDs = "en_welcome_email_id" | "en_email_heading_id" |
"en_footer_title_id" | "en_footer_sendoff_id" | "ja_welcome_email_id" |
"ja_email_heading_id" | "ja_footer_title_id" | "ja_footer_sendoff_id" |
"pt_welcome_email_id" | "pt_email_heading_id" | "pt_footer_title_id" |
"pt_footer_sendoff_id"Try

```

We generally recommend that people use ahead-of-time generation for large string unions, but this is useful in smaller cases.

### String Unions in Types

The power in template literals comes when defining a new string based on information inside a type.

Consider the case where a function (makeWatchedObject) adds a new function called on() to a passed object. In JavaScript, its call might look like: makeWatchedObject(baseObject). We can imagine the base object as looking like:

```

tsconst passedObject = { firstName: "Saoirse", lastName: "Ronan", age: 26,};Try

```

The on function that will be added to the base object expects two arguments, an eventName (a string) and a callback (a function).

The eventName should be of the form attributeInThePassedObject + "Changed"; thus, firstNameChanged as derived from the attribute firstName in the base object.

The callback function, when called:

Should be passed a value of the type associated with the name

attributeInThePassedObject; thus, since firstName is typed as string, the callback for

the firstNameChanged event expects a string to be passed to it at call time. Similarly events associated with age should expect to be called with a number argument  
Should have void return type (for simplicity of demonstration)

The naive function signature of on() might thus be: on(eventName: string, callback: (newValue: any) => void). However, in the preceding description, we identified important type constraints that we'd like to document in our code. Template Literal types let us bring these constraints into our code.

```
tsconst person = makeWatchedObject({ firstName: "Saoirse", lastName: "Ronan", age: 26}); // makeWatchedObject has added `on` to the anonymous Object
person.on("firstNameChanged", (newValue) => { console.log(`firstName was changed to ${newValue}!`)}); Try
```

Notice that on listens on the event "firstNameChanged", not just "firstName". Our naive specification of on() could be made more robust if we were to ensure that the set of eligible event names was constrained by the union of attribute names in the watched object with "Changed" added at the end. While we are comfortable with doing such a calculation in JavaScript i.e. Object.keys(passedObject).map(x => `\${x}Changed`), template literals inside the type system provide a similar approach to string manipulation:

```
tstype PropEventSource<Type> = { on(eventName: `${string & keyof Type}Changed`, callback: (newValue: any) => void): void; }; // Create a "watched object" with an `on` method
// so that you can watch for changes to properties. declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>; Try
```

With this, we can build something that errors when given the wrong property:

```
tsconst person = makeWatchedObject({ firstName: "Saoirse", lastName: "Ronan", age: 26});
person.on("firstNameChanged", () => {}); // Prevent easy human error (using the key instead of the event name)
person.on("firstName", () => {}); Argument of type "firstName" is not assignable to parameter of type "firstNameChanged" |
```

```
"lastNameChanged" | "ageChanged".2345
Argument of type "firstName" is not assignable to parameter of type "firstNameChanged" | "lastNameChanged" | "ageChanged". // It's typo-resistant
person.on("frstNameChanged", () => {}); Argument of type "frstNameChanged" is not assignable to parameter of type "firstNameChanged" | "lastNameChanged" | "ageChanged".2345
Argument of type "frstNameChanged" is not assignable to parameter of type "firstNameChanged" | "lastNameChanged" | "ageChanged". Try
```

Inference with Template Literals

Notice that we did not benefit from all the information provided in the original passed object. Given change of a firstName (i.e. a firstNameChanged event), we should expect that the callback will receive an argument of type string. Similarly, the callback for a change to age should receive a number argument. We're naively using any to type the callback's argument. Again, template literal types make it possible to ensure an attribute's data type will be the same type as that attribute's callback's first argument. The key insight that makes this possible is this: we can use a function with a generic such that:

The literal used in the first argument is captured as a literal type

That literal type can be validated as being in the union of valid attributes in the generic  
The type of the validated attribute can be looked up in the generic's structure using  
Indexed Access

This typing information can then be applied to ensure the argument to the  
callback function is of the same type

```
tstype PropEventSource<Type> = { on<Key extends string & keyof Type>  
(eventName: `${Key}Changed`, callback: (newValue: Type[Key]) => void): void;}; declare  
function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>; const  
person = makeWatchedObject({ firstName: "Saoirse", lastName: "Ronan", age:  
26}); person.on("firstNameChanged", newName => { (parameter)  
newName: string console.log(`new name is ${newName.toUpperCase()}  
`);}); person.on("ageChanged", newAge => { (parameter) newAge:  
number if (newAge < 0) { console.warn("warning! negative age"); })})Try
```

Here we made on into a generic method.

When a user calls with the string "firstNameChanged", TypeScript will try to infer the  
right type for Key.

To do that, it will match Key against the content before "Changed" and infer the string  
"firstName".

Once TypeScript figures that out, the on method can fetch the type of firstName on the  
original object, which is string in this case.

Similarly, when called with "ageChanged", TypeScript finds the type for the property  
age which is number.

Inference can be combined in different ways, often to deconstruct strings, and  
reconstruct them in different ways.

Intrinsic String Manipulation Types

To help with string manipulation, TypeScript includes a set of types which can be used  
in string manipulation. These types come built-in to the compiler for performance and  
can't be found in the .d.ts files included with TypeScript.

Uppercase<StringType>

Converts each character in the string to the uppercase version.

Example

```
tstype Greeting = "Hello, world" type ShoutyGreeting = Uppercase<Greeting>  
type ShoutyGreeting = "HELLO, WORLD" type ASCIIKey<Str extends string> =  
`ID-${Uppercase<Str>}` type MainID = ASCIIKey<"my_app"> type MainID =  
"ID-MY_APP"Try
```

Lowercase<StringType>

Converts each character in the string to the lowercase equivalent.

Example

```
tstype Greeting = "Hello, world" type QuietGreeting = Lowercase<Greeting> type  
QuietGreeting = "hello, world" type ASCIIKey<Str extends string> = `id-  
${Lowercase<Str>}` type MainID = ASCIIKey<"MY_APP"> type MainID = "id-  
my_app"Try
```

Capitalize<StringType>

Converts the first character in the string to an uppercase equivalent.

Example

```
type LowercaseGreeting = "hello, world"; type Greeting =  
Capitalize<LowercaseGreeting>; type Greeting = "Hello, world"Try  
Uncapitalize<StringType>
```

Converts the first character in the string to a lowercase equivalent.

Example

```
type UppercaseGreeting = "HELLO WORLD"; type UncomfortableGreeting =  
Uncapitalize<UppercaseGreeting>; type UncomfortableGreeting = "hELLO  
WORLD"Try
```

Technical details on the intrinsic string manipulation types

The code, as of TypeScript 4.1, for these intrinsic functions uses the JavaScript string runtime functions directly for manipulation and are not locale aware.

```
function applyStringMapping(symbol: Symbol, str: string) {  
  switch (intrinsicTypeKinds.get(symbol.escapedName as string)) {  
    case IntrinsicTypeKind.Uppercase: return str.toUpperCase();  
    case IntrinsicTypeKind.Lowercase: return str.toLowerCase();  
    case IntrinsicTypeKind.Capitalize: return str.charAt(0).toUpperCase() + str.slice(1);  
    case IntrinsicTypeKind.Uncapitalize: return str.charAt(0).toLowerCase() +  
str.slice(1);  
  }  
  return str;  
}
```