CHAPTER 5

# Working with any

Type systems were traditionally binary affairs: either a language had a fully static type system or a fully dynamic one. TypeScript blurs the line, because its type system is *optional* and *gradual*. You can add types to parts of your program but not others.

This is essential for migrating existing JavaScript codebases to TypeScript bit by bit (Chapter 8). Key to this is the `any` type, which effectively disables type checking for parts of your code. It is both powerful and prone to abuse. Learning to use `any` wisely is essential for writing effective TypeScript. This chapter walks you through how to limit the downsides of `any` while still retaining its benefits.

## Item 38: Use the Narrowest Possible Scope for any Types

Consider this code:

```
function processBar(b: Bar) { /* ... */ }

function f() {
  const x = expressionReturningFoo();
  processBar(x);
  //         ~ Argument of type 'Foo' is not assignable to
  //           parameter of type 'Bar'
}
```

If you somehow know from context that `x` is assignable to `Bar` in addition to `Foo`, you can force TypeScript to accept this code in two ways:

```
function f1() {
  const x: any = expressionReturningFoo();  // Don't do this
  processBar(x);
}

function f2() {
```

```
  const x = expressionReturningFoo();
  processBar(x as any);  // Prefer this
}
```

Of these, the second form is vastly preferable. Why? Because the `any` type is scoped to a single expression in a function argument. It has no effect outside this argument or this line. If code after the `processBar` call references `x`, its type will still be `Foo`, and it will still be able to trigger type errors, whereas in the first example its type is `any` until it goes out of scope at the end of the function.

The stakes become significantly higher if you *return* `x` from this function. Look what happens:

```
function f1() {
  const x: any = expressionReturningFoo();
  processBar(x);
  return x;
}

function g() {
  const foo = f1();  // Type is any
  foo.fooMethod();  // This call is unchecked!
}
```

An `any` return type is "contagious" in that it can spread throughout a codebase. As a result of our changes to `f`, an `any` type has quietly appeared in `g`. This would not have happened with the more narrowly scoped `any` in `f2`.

(This is a good reason to consider including explicit return type annotations, even when the return type can be inferred. It prevents an `any` type from "escaping." See discussion in Item 19.)

We used `any` here to silence an error that we believed to be incorrect. Another way to do this is with `@ts-ignore`:

```
function f1() {
  const x = expressionReturningFoo();
  // @ts-ignore
  processBar(x);
  return x;
}
```

This silences an error on the next line, leaving the type of `x` unchanged. Try not to lean too heavily on `@ts-ignore`: the type checker usually has a good reason to complain. It also means that if the error on the next line changes to something more problematic, you won't know.

You may also run into situations where you get a type error for just one property in a larger object:

```
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value
 // ~~~ Property ... missing in type 'Bar' but required in type 'Foo'
  }
};
```

You can silence errors like this by throwing an `as any` around the whole `config` object:

```
const config: Config = {
  a: 1,
  b: 2,
  c: {
    key: value
  }
} as any;  // Don't do this!
```

But this has the side effect of disabling type checking for the other properties (`a` and `b`) as well. Using a more narrowly scoped `any` limits the damage:

```
const config: Config = {
  a: 1,
  b: 2,  // These properties are still checked
  c: {
    key: value as any
  }
};
```

## Things to Remember

- Make your uses of `any` as narrowly scoped as possible to avoid undesired loss of type safety elsewhere in your code.
- Never return an `any` type from a function. This will silently lead to the loss of type safety for any client calling the function.
- Consider `@ts-ignore` as an alternative to `any` if you need to silence one error.

# Item 39: Prefer More Precise Variants of any to Plain any

The `any` type encompasses all values that can be expressed in JavaScript. This is a vast set! It includes not just all numbers and strings, but all arrays, objects, regular expressions, functions, classes, and DOM elements, not to mention `null` and `undefined`. When you use an `any` type, ask whether you really had something more specific in mind. Would it be OK to pass in a regular expression or a function?

Often the answer is "no," in which case you might be able to retain some type safety by using a more specific type:

```
function getLengthBad(array: any) {  // Don't do this!
  return array.length;
}

function getLength(array: any[]) {
  return array.length;
}
```

The latter version, which uses `any[]` instead of `any`, is better in three ways:

- The reference to `array.length` in the function body is type checked.
- The function's return type is inferred as `number` instead of `any`.
- Calls to `getLength` will be checked to ensure that the parameter is an array:

```
getLengthBad(/123/);  // No error, returns undefined
getLength(/123/);
       // ~~~~~ Argument of type 'RegExp' is not assignable
       //       to parameter of type 'any[]'
```

If you expect a parameter to be an array of arrays but don't care about the type, you can use `any[][]`. If you expect some sort of object but don't know what the values will be, you can use `{[key: string]: any}`:

```
function hasTwelveLetterKey(o: {[key: string]: any}) {
  for (const key in o) {
    if (key.length === 12) {
      return true;
    }
  }
  return false;
}
```

You could also use the `object` type in this situation, which includes all non-primitive types. This is slightly different in that, while you can still enumerate keys, you can't access the values of any of them:

```
function hasTwelveLetterKey(o: object) {
  for (const key in o) {
    if (key.length === 12) {
      console.log(key, o[key]);
                  //  ~~~~~~ Element implicitly has an 'any' type
                  //         because type '{}' has no index signature
      return true;
    }
  }
  return false;
}
```

If this sort of type fits your needs, you might also be interested in the `unknown` type. See Item 42.

Avoid using `any` if you expect a function type. You have several options here depending on how specific you want to get:

```
type Fn0 = () => any;  // any function callable with no params
type Fn1 = (arg: any) => any;  // With one param
type FnN = (...args: any[]) => any;  // With any number of params
                                     // same as "Function" type
```

All of these are more precise than `any` and hence preferable to it. Note the use of `any[]` as the type for the rest parameter in the last example. `any` would also work here but would be less precise:

```
const numArgsBad = (...args: any) => args.length; // Returns any
const numArgsGood = (...args: any[]) => args.length;  // Returns number
```

This is perhaps the most common use of the `any[]` type.

### Things to Remember

- When you use `any`, think about whether any JavaScript value is truly permissible.
- Prefer more precise forms of `any` such as `any[]` or `{[id: string]: any}` or `() => any` if they more accurately model your data.

## Item 40: Hide Unsafe Type Assertions in Well-Typed Functions

There are many functions whose type signatures are easy to write but whose implementations are quite difficult to write in type-safe code. And while writing type-safe implementations is a noble goal, it may not be worth the difficulty to deal with edge cases that you know don't come up in your code. If a reasonable attempt at a type-safe implementation doesn't work, use an unsafe type assertion hidden inside a function with the right type signature. Unsafe assertions hidden inside well-typed functions are much better than unsafe assertions scattered throughout your code.

Suppose you want to make a function cache its last call. This is a common technique for eliminating expensive function calls with frameworks like React.[1] It would be nice to write a general `cacheLast` wrapper that adds this behavior to any function. Its declaration is easy to write:

1 If you are using React, you should use the built-in `useMemo` hook, rather than rolling your own.

```
declare function cacheLast<T extends Function>(fn: T): T;
```

Here's an attempt at an implementation:

```
function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[]|null = null;
  let lastResult: any;
  return function(...args: any[]) {
      // ~~~~~~~~~~~~~~~~~~~~~~~~~
      //          Type '(...args: any[]) => any' is not assignable to type 'T'
    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args);
      lastArgs = args;
    }
    return lastResult;
  };
}
```

The error makes sense: TypeScript has no reason to believe that this very loose function has any relation to `T`. But you know that the type system will enforce that it's called with the right parameters and that its return value is given the correct type. So you shouldn't expect too many problems if you add a type assertion here:

```
function cacheLast<T extends Function>(fn: T): T {
  let lastArgs: any[]|null = null;
  let lastResult: any;
  return function(...args: any[]) {
    if (!lastArgs || !shallowEqual(lastArgs, args)) {
      lastResult = fn(...args);
      lastArgs = args;
    }
    return lastResult;
  } as unknown as T;
}
```

And indeed this will work great for any simple function you pass it. There are quite a few `any` types hidden in this implementation, but you've kept them out of the type signature, so the code that calls `cacheLast` will be none the wiser.

(Is this actually safe? There are a few real problems with this implementation: it doesn't check that the values of `this` for successive calls are the same. And if the original function had properties defined on it, then the wrapped function would not have these, so it wouldn't have the same type. But if you know that these situations don't come up in your code, this implementation is just fine. This function *can* be written in a type-safe way, but it is a more complex exercise that is left to the reader.)

The `shallowEqual` function from the previous example operated on two arrays and is easy to type and implement. But the object variation is more interesting. As with `cacheLast`, it's easy to write its type signature:

```
declare function shallowObjectEqual<T extends object>(a: T, b: T): boolean;
```

The implementation requires some care since there's no guarantee that `a` and `b` have the same keys (see Item 54):

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  for (const [k, aVal] of Object.entries(a)) {
    if (!(k in b) || aVal !== b[k]) {
                          // ~~~~ Element implicitly has an 'any' type
                          //      because type '{}' has no index signature
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
}
```

It's a bit surprising that TypeScript complains about the `b[k]` access despite your having just checked that `k in b` is true. But it does, so you have no choice but to cast:

```
function shallowObjectEqual<T extends object>(a: T, b: T): boolean {
  for (const [k, aVal] of Object.entries(a)) {
    if (!(k in b) || aVal !== (b as any)[k]) {
      return false;
    }
  }
  return Object.keys(a).length === Object.keys(b).length;
}
```

This type assertion is harmless (since you've checked `k in b`), and you're left with a correct function with a clear type signature. This is much preferable to scattering iteration and assertions to check for object equality throughout your code!

## Things to Remember

- Sometimes unsafe type assertions are necessary or expedient. When you need to use one, hide it inside a function with a correct signature.

# Item 41: Understand Evolving any

In TypeScript a variable's type is generally determined when it is declared. After this, it can be *refined* (by checking if it is `null`, for instance), but it cannot expand to include new values. There is one notable exception to this, however, involving `any` types.

In JavaScript, you might write a function to generate a range of numbers like this:

```
function range(start, limit) {
  const out = [];
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
```

```
  return out;
}
```

When you convert this to TypeScript, it works exactly as you'd expect:

```
function range(start: number, limit: number) {
  const out = [];
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out;  // Return type inferred as number[]
}
```

Upon closer inspection, however, it's surprising that this works! How does TypeScript know that the type of `out` is `number[]` when it's initialized as `[]`, which could be an array of any type?

Inspecting each of the three occurrences of `out` to reveal its inferred type starts to tell the story:

```
function range(start: number, limit: number) {
  const out = [];  // Type is any[]
  for (let i = start; i < limit; i++) {
    out.push(i);  // Type of out is any[]
  }
  return out;  // Type is number[]
}
```

The type of `out` starts as `any[]`, an undifferentiated array. But as we push `number` values onto it, its type "evolves" to become `number[]`.

This is distinct from narrowing (Item 22). An array's type can expand by pushing different elements onto it:

```
const result = [];  // Type is any[]
result.push('a');
result  // Type is string[]
result.push(1);
result  // Type is (string | number)[]
```

With conditionals, the type can even vary across branches. Here we show the same behavior with a simple value, rather than an array:

```
let val;  // Type is any
if (Math.random() < 0.5) {
  val = /hello/;
  val  // Type is RegExp
} else {
  val = 12;
  val  // Type is number
}
val  // Type is number | RegExp
```

A final case that triggers this "evolving any" behavior is if a variable is initially `null`. This often comes up when you set a value in a `try/catch` block:

```
let val = null;  // Type is any
try {
  somethingDangerous();
  val = 12;
  val  // Type is number
} catch (e) {
  console.warn('alas!');
}
val  // Type is number | null
```

Interestingly, this behavior only happens when a variable's type is implicitly `any` with `noImplicitAny` set! Adding an *explicit* `any` keeps the type constant:

```
let val: any;  // Type is any
if (Math.random() < 0.5) {
  val = /hello/;
  val  // Type is any
} else {
  val = 12;
  val  // Type is any
}
val  // Type is any
```

This behavior can be confusing to follow in your editor since the type is only "evolved" *after* you assign or push an element. Inspecting the type on the line with the assignment will still show `any` or `any[]`.

If you use a value before any assignment to it, you'll get an implicit any error:

```
function range(start: number, limit: number) {
  const out = [];
  //    ~~~ Variable 'out' implicitly has type 'any[]' in some
  //        locations where its type cannot be determined
  if (start === limit) {
    return out;
    //     ~~~ Variable 'out' implicitly has an 'any[]' type
  }
  for (let i = start; i < limit; i++) {
    out.push(i);
  }
  return out;
}
```

Put another way, "evolving" `any` types are only `any` when you *write* to them. If you try to *read* from them while they're still `any`, you'll get an error.

Implicit any types do not evolve through function calls. The arrow function here trips up inference:

```
function makeSquares(start: number, limit: number) {
  const out = [];
     // ~~~ Variable 'out' implicitly has type 'any[]' in some locations
  range(start, limit).forEach(i => {
    out.push(i * i);
  });
  return out;
      // ~~~ Variable 'out' implicitly has an 'any[]' type
}
```

In cases like this, you may want to consider using an array's `map` and `filter` methods to build arrays in a single statement and avoid iteration and evolving `any` entirely. See Items 23 and 27.

Evolving `any` comes with all the usual caveats about type inference. Is the correct type for your array really `(string|number)[]`? Or should it be `number[]` and you incorrectly pushed a `string`? You may still want to provide an explicit type annotation to get better error checking instead of using evolving `any`.

## Things to Remember

- While TypeScript types typically only *refine*, implicit `any` and `any[]` types are allowed to *evolve*. You should be able to recognize and understand this construct where it occurs.
- For better error checking, consider providing an explicit type annotation instead of using evolving `any`.

# Item 42: Use unknown Instead of any for Values with an Unknown Type

Suppose you want to write a YAML parser (YAML can represent the same set of values as JSON but allows a superset of JSON's syntax). What should the return type of your `parseYAML` method be? It's tempting to make it `any` (like `JSON.parse`):

```
function parseYAML(yaml: string): any {
  // ...
}
```

But this flies in the face of Item 38's advice to avoid "contagious" `any` types, specifically by not returning them from functions.

Ideally you'd like your users to immediately assign the result to another type:

```
interface Book {
  name: string;
  author: string;
}
const book: Book = parseYAML(`
  name: Wuthering Heights
  author: Emily Brontë
`);
```

Without the type declarations, though, the `book` variable would quietly get an `any` type, thwarting type checking wherever it's used:

```
const book = parseYAML(`
  name: Jane Eyre
  author: Charlotte Brontë
`);
alert(book.title);  // No error, alerts "undefined" at runtime
book('read');  // No error, throws "TypeError: book is not a
               // function" at runtime
```

A safer alternative would be to have `parseYAML` return an `unknown` type:

```
function safeParseYAML(yaml: string): unknown {
  return parseYAML(yaml);
}
const book = safeParseYAML(`
  name: The Tenant of Wildfell Hall
  author: Anne Brontë
`);
alert(book.title);
   // ~~~~ Object is of type 'unknown'
book("read");
// ~~~~~~~~~~ Object is of type 'unknown'
```

To understand the `unknown` type, it helps to think about `any` in terms of assignability. The power and danger of `any` come from two properties:

- Any type is assignable to the `any` type.
- The `any` type is assignable to any other type.[2]

In the context of "thinking of types as sets of values" (Item 7), `any` clearly doesn't fit into the type system, since a set can't simultaneously be both a subset and a superset of all other sets. This is the source of `any`'s power but also the reason it's problematic. Since the type checker is set-based, the use of `any` effectively disables it.

The `unknown` type is an alternative to `any` that *does* fit into the type system. It has the first property (any type is assignable to `unknown`) but not the second (`unknown` is only

2 With the exception of `never`.

assignable to `unknown` and, of course, `any`). The `never` type is the opposite: it has the second property (can be assigned to any other type) but not the first (nothing can be assigned to `never`).

Attempting to access a property on a value with the `unknown` type is an error. So is attempting to call it or do arithmetic with it. You can't do much with `unknown`, which is exactly the point. The errors about an `unknown` type will encourage you to add an appropriate type:

```
const book = safeParseYAML(`
  name: Villette
  author: Charlotte Brontë
`) as Book;
alert(book.title);
       // ~~~~~ Property 'title' does not exist on type 'Book'
book('read');
// ~~~~~~~~~ this expression is not callable
```

These errors are more sensible. Since `unknown` is not assignable to other types, a type assertion is required. But it is also appropriate: we really do know more about the type of the resulting object than TypeScript does.

`unknown` is appropriate whenever you know that there will be a value but you don't know its type. The result of `parseYAML` is one example, but there are others. In the GeoJSON spec, for example, the `properties` property of a Feature is a grab-bag of anything JSON serializable. So `unknown` makes sense:

```
interface Feature {
  id?: string | number;
  geometry: Geometry;
  properties: unknown;
}
```

A type assertion isn't the only way to recover a type from an `unknown` object. An `instanceof` check will do:

```
function processValue(val: unknown) {
  if (val instanceof Date) {
    val  // Type is Date
  }
}
```

You can also use a user-defined type guard:

```
function isBook(val: unknown): val is Book {
  return (
      typeof(val) === 'object' && val !== null &&
      'name' in val && 'author' in val
  );
}
function processValue(val: unknown) {
```

```
    if (isBook(val)) {
      val;  // Type is Book
    }
  }
```

TypeScript requires quite a bit of proof to narrow an `unknown` type: in order to avoid errors on the `in` checks, you first have to demonstrate that `val` is an object type and that it is non-`null` (since `typeof null === 'object'`).

You'll sometimes see a generic parameter used instead of `unknown`. You could have declared the `safeParseYAML` function this way:

```
function safeParseYAML<T>(yaml: string): T {
  return parseYAML(yaml);
}
```

This is generally considered bad style in TypeScript, however. It looks different than a type assertion, but is functionally the same. Better to just return `unknown` and force your users to use an assertion or narrow to the type they want.

`unknown` can also be used instead of `any` in "double assertions":

```
declare const foo: Foo;
let barAny = foo as any as Bar;
let barUnk = foo as unknown as Bar;
```

These are functionally equivalent, but the `unknown` form has less risk if you do a refactor and break up the two assertions. In that case the `any` could escape and spread. If the `unknown` type escapes, it will probably just produce an error.

As a final note, you may see code that uses `object` or `{}` in a similar way to how `unknown` has been described in this item. They are also broad types but are slightly narrower than `unknown`:

- The `{}` type consists of all values except `null` and `undefined`.
- The `object` type consists of all non-primitive types. This doesn't include `true` or `12` or `"foo"` but does include objects and arrays.

The use of `{}` was more common before the `unknown` type was introduced. Uses today are somewhat rare: only use `{}` instead of `unknown` if you really do know that `null` and `undefined` aren't possibilities.

## Things to Remember

- The `unknown` type is a type-safe alternative to `any`. Use it when you know you have a value but do not know what its type is.
- Use `unknown` to force your users to use a type assertion or do type checking.

- Understand the difference between `{}`, `object`, and `unknown`.

## Item 43: Prefer Type-Safe Approaches to Monkey Patching

One of the most famous features of JavaScript is that its objects and classes are "open" in the sense that you can add arbitrary properties to them. This is occasionally used to create global variables on web pages by assigning to `window` or `document`:

```
window.monkey = 'Tamarin';
document.monkey = 'Howler';
```

or to attach data to DOM elements:

```
const el = document.getElementById('colobus');
el.home = 'tree';
```

This style is particularly common with code that uses jQuery.

You can even attach properties to the prototypes of built-ins, with sometimes surprising results:

```
>  RegExp.prototype.monkey = 'Capuchin'
"Capuchin"
> /123/.monkey
"Capuchin"
```

These approaches are generally not good designs. When you attach data to `window` or a DOM node, you are essentially turning it into a global variable. This makes it easy to inadvertently introduce dependencies between far-flung parts of your program and means that you have to think about side effects whenever you call a function.

Adding TypeScript introduces another problem: while the type checker knows about built-in properties of `Document` and `HTMLElement`, it certainly doesn't know about the ones you've added:

```
document.monkey = 'Tamarin';
      // ~~~~~~ Property 'monkey' does not exist on type 'Document'
```

The most straightforward way to fix this error is with an `any` assertion:

```
(document as any).monkey = 'Tamarin';  // OK
```

This satisfies the type checker, but, as should be no surprise by now, it has some downsides. As with any use of `any`, you lose type safety and language services:

```
(document as any).monky = 'Tamarin';  // Also OK, misspelled
(document as any).monkey = /Tamarin/;  // Also OK, wrong type
```

The best solution is to move your data out of `document` or the DOM. But if you can't (perhaps you're using a library that requires it or are in the process of migrating a JavaScript application), then you have a few next-best options available.

One is to use an augmentation, one of the special abilities of `interface` (Item 13):

```
interface Document {
  /** Genus or species of monkey patch */
  monkey: string;
}

document.monkey = 'Tamarin';  // OK
```

This is an improvement over using `any` in a few ways:

- You get type safety. The type checker will flag misspellings or assignments of the wrong type.
- You can attach documentation to the property (Item 48).
- You get autocomplete on the property.
- There is a record of precisely what the monkey patch is.

In a module context (i.e., a TypeScript file that uses `import` / `export`), you'll need to add a `declare global` to make this work:

```
export {};
declare global {
  interface Document {
    /** Genus or species of monkey patch */
    monkey: string;
  }
}
document.monkey = 'Tamarin';  // OK
```

The main issues with using an augmentation have to do with scope. First, the augmentation applies globally. You can't hide it from other parts of your code or from libraries. And second, if you assign the property while your application is running, there's no way to introduce the augmentation only after this has happened. This is particularly problematic when you patch HTML Elements, where some elements on the page will have the property and some will not. For this reason, you might want to declare the property to be `string|undefined`. This is more accurate, but will make the type less convenient to work with.

Another approach is to use a more precise type assertion:

```
interface MonkeyDocument extends Document {
  /** Genus or species of monkey patch */
  monkey: string;
}
```

```
(document as MonkeyDocument).monkey = 'Macaque';
```

TypeScript is OK with the type assertion because `Document` and `MonkeyDocument` share properties (Item 9). And you get type safety in the assignment. The scope issues are also more manageable: there's no global modification of the `Document` type, just the introduction of a new type (which is only in scope if you import it). You have to write an assertion (or introduce a new variable) whenever you reference the monkey-patched property. But you can take that as encouragement to refactor into something more structured. Monkey patching shouldn't be *too* easy!

### Things to Remember

- Prefer structured code to storing data in globals or on the DOM.
- If you must store data on built-in types, use one of the type-safe approaches (augmentation or asserting a custom interface).
- Understand the scoping issues of augmentations.

## Item 44: Track Your Type Coverage to Prevent Regressions in Type Safety

Are you safe from the problems associated with any types once you've added type annotations for values with implicit `any` types and enabled `noImplicitAny`? The answer is "no"; `any` types can still enter your program in two main ways:

*Explicit* `any` *types*
: Even if you follow the advice of Items 38 and 39, making your `any` types both narrow and specific, they remain `any` types. In particular, types like `any[]` and `{[key: string]: any}` become plain `any`s once you index into them, and the resulting `any` types can flow through your code.

*From third-party type declarations*
: This is particularly insidious since `any` types from an `@types` declaration file enter silently: even though you have `noImplicitAny` enabled and you never typed `any`, you still have `any` types flowing through your code.

Because of the negative effects `any` types can have on type safety and developer experience (Item 5), it's a good idea to keep track of the number of them in your codebase. There are many ways to do this, including the `type-coverage` package on npm:

```
$ npx type-coverage
9985 / 10117 98.69%
```

This means that, of the 10,117 symbols in this project, 9,985 (98.69%) had a type other than `any` or an alias to `any`. If a change inadvertently introduces an `any` type and it flows through your code, you'll see a corresponding drop in this percentage.

In some ways this percentage is a way of keeping score on how well you've followed the advice of the other items in this chapter. Using narrowly scoped `any` will reduce the number of symbols with `any` types, and so will using more specific forms like `any[]`. Tracking this numerically helps you make sure things only get better over time.

Even collecting type coverage information once can be informative. Running `type-coverage` with the `--detail` flag will print where every `any` type occurs in your code:

```
$ npx type-coverage --detail
path/to/code.ts:1:10 getColumnInfo
path/to/module.ts:7:1 pt2
...
```

These are worth investigating because they're likely to turn up sources of `any`s that you hadn't considered. Let's look at a few examples.

Explicit `any` types are often the result of choices you made for expediency earlier on. Perhaps you were getting a type error that you didn't want to take the time to sort out. Or maybe the type was one that you hadn't written out yet. Or you might have just been in a rush.

Type assertions with `any` can prevent types from flowing where they otherwise would. Perhaps you've built an application that works with tabular data and needed a single-parameter function that built up some kind of column description:

```
function getColumnInfo(name: string): any {
  return utils.buildColumnInfo(appState.dataSchema, name);  // Returns any
}
```

The `utils.buildColumnInfo` function returned `any` at some point. As a reminder, you added a comment and an explicit ": any" annotation to the function.

However, in the intervening months you've also added a type for `ColumnInfo`, and `utils.buildColumnInfo` no longer returns `any`. The `any` annotation is now throwing away valuable type information. Get rid of it!

Third-party `any` types can come in a few forms, but the most extreme is when you give an entire module an `any` type:

```
declare module 'my-module';
```

Now you can import anything from `my-module` without error. These symbols all have `any` types and will lead to more `any` types if you pass values through them:

```
import {someMethod, someSymbol} from 'my-module';  // OK

const pt1 = {
  x: 1,
  y: 2,
};  // type is {x: number, y: number}
const pt2 = someMethod(pt1, someSymbol);  // OK, pt2's type is any
```

Since the usage looks identical to a well-typed module, it's easy to forget that you stubbed out the module. Or maybe a coworker did it and you never knew in the first place. It's worth revisiting these from time to time. Maybe there are official type declarations for the module. Or perhaps you've gained enough understanding of the module to write types yourself and contribute them back to the community.

Another common source of `any`s with third-party declarations is when there's a bug in the types. Maybe the declarations didn't follow the advice of Item 29 and declared a function to return a union type when in fact it returns something much more specific. When you first used the function this didn't seem worth fixing so you used an `any` assertion. But maybe the declarations have been fixed since then. Or maybe it's time to fix them yourself!

The considerations that led you to use an `any` type might not apply any more. Maybe there's a type you can plug in now where previously you used `any`. Maybe an unsafe type assertion is no longer necessary. Maybe the bug in the type declarations you were working around has been fixed. Tracking your type coverage highlights these choices and encourages you to keep revisiting them.

## Things to Remember

- Even with `noImplicitAny` set, `any` types can make their way into your code either through explicit `any`s or third-party type declarations (`@types`).
- Consider tracking how well-typed your program is. This will encourage you to revisit decisions about using `any` and increase type safety over time.