CHAPTER 3
Type Inference

For programming languages used in industry, “statically typed” and “explicitly typed” have traditionally been synonymous. C, C++, Java: they all made you write out your types. But academic languages never conflated these two things: languages like ML and Haskell have long had sophisticated type inference systems, and in the past decade this has begun to work its way into industry languages. C++ has added `auto`, and Java has added `var`.

TypeScript makes extensive use of type inference. Used well, this can dramatically reduce the number of type annotations your code requires to get full type safety. One of the easiest ways to tell a TypeScript beginner from a more experienced user is by the number of type annotations. An experienced TypeScript developer will use relatively few annotations (but use them to great effect), while a beginner may drown their code in redundant type annotations.

This chapter shows you some of the problems that can arise with type inference and how to fix them. After reading it, you should have a good understanding of how TypeScript infers types, when you still need to write type declarations, and when it’s a good idea to write type declarations even when a type can be inferred.

Item 19: Avoid Cluttering Your Code with Inferable Types

The first thing that many new TypeScript developers do when they convert a codebase from JavaScript is fill it with type annotations. TypeScript is about *types*, after all! But in TypeScript many annotations are unnecessary. Declaring types for all your variables is counterproductive and is considered poor style.

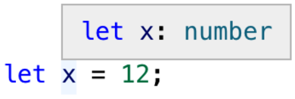
Don’t write:

```
let x: number = 12;
```

Instead, just write:

```
let x = 12;
```

If you mouse over x in your editor, you'll see that its type has been inferred as number (as shown in Figure 3-1).



```
let x: number
let x = 12;
```

Figure 3-1. A text editor showing that the inferred type of x is number.

The explicit type annotation is redundant. Writing it just adds noise. If you're unsure of the type, you can check it in your editor.

TypeScript will also infer the types of more complex objects. Instead of:

```
const person: {
  name: string;
  born: {
    where: string;
    when: string;
  };
  died: {
    where: string;
    when: string;
  }
} = {
  name: 'Sojourner Truth',
  born: {
    where: 'Swartekill, NY',
    when: 'c.1797',
  },
  died: {
    where: 'Battle Creek, MI',
    when: 'Nov. 26, 1883'
  }
};
```

you can just write:

```
const person = {
  name: 'Sojourner Truth',
  born: {
    where: 'Swartekill, NY',
    when: 'c.1797',
  },
  died: {
    where: 'Battle Creek, MI',
    when: 'Nov. 26, 1883'
  }
};
```

Again, the types are exactly the same. Writing the type in addition to the value just adds noise here. (Item 21 has more to say on the types inferred for object literals.)

What's true for objects is also true for arrays. TypeScript has no trouble figuring out the return type of this function based on its inputs and operations:

```
function square(nums: number[]) {
  return nums.map(x => x * x);
}
const squares = square([1, 2, 3, 4]); // Type is number[]
```

TypeScript may infer something more precise than what you expected. This is generally a good thing. For example:

```
const axis1: string = 'x';  // Type is string
const axis2 = 'y';  // Type is "y"
```

`"y"` is a more precise type for the `axis` variable. Item 21 gives an example of how this can fix a type error.

Allowing types to be inferred can also facilitate refactoring. Say you have a `Product` type and a function to log it:

```
interface Product {
  id: number;
  name: string;
  price: number;
}

function logProduct(product: Product) {
  const id: number = product.id;
  const name: string = product.name;
  const price: number = product.price;
  console.log(id, name, price);
}
```

At some point you learn that product IDs might have letters in them in addition to numbers. So you change the type of `id` in `Product`. Because you included explicit annotations on all the variables in `logProduct`, this produces an error:

```
interface Product {
  id: string;
  name: string;
  price: number;
}

function logProduct(product: Product) {
  const id: number = product.id;
     // ~~ Type 'string' is not assignable to type 'number'
  const name: string = product.name;
  const price: number = product.price;
  console.log(id, name, price);
}
```

Had you left off all the annotations in the `logProduct` function body, the code would have passed the type checker without modification.

A better implementation of `logProduct` would use destructuring assignment (Item 58):

```
function logProduct(product: Product) {
  const {id, name, price} = product;
  console.log(id, name, price);
}
```

This version allows the types of all the local variables to be inferred. The corresponding version with explicit type annotations is repetitive and cluttered:

```
function logProduct(product: Product) {
  const {id, name, price}: {id: string; name: string; price: number } = product;
  console.log(id, name, price);
}
```

Explicit type annotations are still required in some situations where TypeScript doesn't have enough context to determine a type on its own. You have seen one of these before: function parameters.

Some languages will infer types for parameters based on their eventual usage, but TypeScript does not. In TypeScript, a variable's type is generally determined when it is first introduced.

Ideal TypeScript code includes type annotations for function/method signatures but not for the local variables created in their bodies. This keeps noise to a minimum and lets readers focus on the implementation logic.

There are some situations where you can leave the type annotations off of function parameters, too. When there's a default value, for example:

```
function parseNumber(str: string, base=10) {
  // ...
}
```

Here the type of `base` is inferred as `number` because of the default value of `10`.

Parameter types can usually be inferred when the function is used as a callback for a library with type declarations. The declarations on `request` and `response` in this example using the express HTTP server library are not required:

```
// Don't do this:
app.get('/health', (request: express.Request, response: express.Response) => {
  response.send('OK');
});
```

```
// Do this:
app.get('/health', (request, response) => {
  response.send('OK');
});
```

Item 26 goes into more depth on how context is used in type inference.

There are a few situations where you may still want to specify a type even where it can be inferred.

One is when you define an object literal:

```
const elmo: Product = {
  name: 'Tickle Me Elmo',
  id: '048188 627152',
  price: 28.99,
};
```

When you specify a type on a definition like this, you enable excess property checking (Item 11). This can help catch errors, particularly for types with optional fields.

You also increase the odds that an error will be reported in the right place. If you leave off the annotation, a mistake in the object's definition will result in a type error where it's used, rather than where it's defined:

```
const furby = {
  name: 'Furby',
  id: 630509430963,
  price: 35,
};
logProduct(furby);
        // ~~~~~ Argument .. is not assignable to parameter of type 'Product'
        //         Types of property 'id' are incompatible
        //         Type 'number' is not assignable to type 'string'
```

With an annotation, you get a more concise error in the place where the mistake was made:

```
 const furby: Product = {
   name: 'Furby',
   id: 630509430963,
// ~~ Type 'number' is not assignable to type 'string'
   price: 35,
 };
 logProduct(furby);
```

Similar considerations apply to a function's return type. You may still want to annotate this even when it can be inferred to ensure that implementation errors don't leak out into uses of the function.

Say you have a function which retrieves a stock quote:

```
function getQuote(ticker: string) {
  return fetch(`https://quotes.example.com/?q=${ticker}`)
```

```
        .then(response => response.json());
}
```

You decide to add a cache to avoid duplicating network requests:

```
const cache: {[ticker: string]: number} = {};
function getQuote(ticker: string) {
  if (ticker in cache) {
    return cache[ticker];
  }
  return fetch(`https://quotes.example.com/?q=${ticker}`)
      .then(response => response.json())
      .then(quote => {
        cache[ticker] = quote;
        return quote;
      });
}
```

There's a mistake in this implementation: you should really be returning `Promise.resolve(cache[ticker])` so that `getQuote` always returns a Promise. The mistake will most likely produce an error…but in the code that calls `getQuote`, rather than in `getQuote` itself:

```
getQuote('MSFT').then(considerBuying);
              // ~~~~ Property 'then' does not exist on type
              //        'number | Promise<any>'
              //      Property 'then' does not exist on type 'number'
```

Had you annotated the intended return type (`Promise<number>`), the error would have been reported in the correct place:

```
const cache: {[ticker: string]: number} = {};
function getQuote(ticker: string): Promise<number> {
  if (ticker in cache) {
    return cache[ticker];
        // ~~~~~~~~~~~~~ Type 'number' is not assignable to 'Promise<number>'
  }
  // ...
}
```

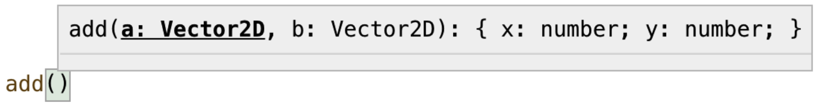
When you annotate the return type, it keeps implementation errors from manifesting as errors in user code. (See Item 25 for a discussion of async functions, which are an effective way to avoid this specific error with Promises.)

Writing out the return type may also help you think more clearly about your function: you should know what its input and output types are *before you implement it*. While the implementation may shift around a bit, the function's contract (its type signature) generally should not. This is similar in spirit to test-driven development (TDD), in which you write the tests that exercise a function before you implement it. Writing the full type signature first helps get you the function you want, rather than the one the implementation makes expedient.

A final reason to annotate return values is if you want to use a named type. You might choose not to write a return type for this function, for example:

```
interface Vector2D { x: number; y: number; }
function add(a: Vector2D, b: Vector2D) {
  return { x: a.x + b.x, y: a.y + b.y };
}
```

TypeScript infers the return type as `{ x: number; y: number; }`. This is compatible with `Vector2D`, but it may be surprising to users of your code when they see `Vector2D` as a type of the input and not of the output (as shown in Figure 3-2).



```
add(a: Vector2D, b: Vector2D): { x: number; y: number; }
add()
```

Figure 3-2. The parameters to the add function have named types, while the inferred return value does not.

If you annotate the return type, the presentation is more straightforward. And if you've written documentation on the type (Item 48) then it will be associated with the returned value as well. As the complexity of the inferred return type increases, it becomes increasingly helpful to provide a name.

If you are using a linter, the eslint rule `no-inferrable-types` (note the variant spelling) can help ensure that all your type annotations are really necessary.

Things to Remember

- Avoid writing type annotations when TypeScript can infer the same type.
- Ideally your code has type annotations in function/method signatures but not on local variables in their bodies.
- Consider using explicit annotations for object literals and function return types even when they can be inferred. This will help prevent implementation errors from surfacing in user code.

Item 20: Use Different Variables for Different Types

In JavaScript it's no problem to reuse a variable to hold a differently typed value for a different purpose:

```
let id = "12-34-56";
fetchProduct(id);  // Expects a string
```

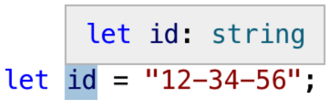
```
id = 123456;
fetchProductBySerialNumber(id);  // Expects a number
```

In TypeScript, this results in two errors:

```
    let id = "12-34-56";
    fetchProduct(id);

    id = 123456;
// ~~ '123456' is not assignable to type 'string'.
    fetchProductBySerialNumber(id);
                                // ~~ Argument of type 'string' is not assignable to
                                //    parameter of type 'number'
```

Hovering over the first `id` in your editor gives a hint as to what's going on (see Figure 3-3).



```
let id: string
let id = "12-34-56";
```

Figure 3-3. The inferred type of id is string.

Based on the value `"12-34-56"`, TypeScript has inferred `id`'s type as `string`. You can't assign a `number` to a `string` and hence the error.

This leads us to a key insight about variables in TypeScript: *while a variable's value can change, its type generally does not*. The one common way a type can change is to narrow (Item 22), but this involves a type getting smaller, not expanding to include new values. There are some important exceptions to this rule (Item 41), but they are the exceptions and not the rule.

How can you use this idea to fix the example? In order for `id`'s type to not change, it must be broad enough to encompass both `string`s and `number`s. This is the very definition of the union type, `string|number`:

```
let id: string|number = "12-34-56";
fetchProduct(id);

id = 123456;  // OK
fetchProductBySerialNumber(id);  // OK
```

This fixes the errors. It's interesting that TypeScript has been able to determine that `id` is really a `string` in the first call and really a `number` in the second. It has narrowed the union type based on the assignment.

While a union type does work, it may create more issues down the road. Union types are harder to work with than simple types like `string` or `number` because you usually have to check what they are before you do anything with them.

The better solution is to introduce a new variable:

```
const id = "12-34-56";
fetchProduct(id);

const serial = 123456;  // OK
fetchProductBySerialNumber(serial);  // OK
```

In the previous version, the first and second `id` were not semantically related to one another. They were only related by the fact that you reused a variable. This was confusing for the type checker and would be confusing for a human reader, too.

The version with two variables is better for a number of reasons:

- It disentangles two unrelated concepts (ID and serial number).
- It allows you to use more specific variable names.
- It improves type inference. No type annotations are needed.
- It results in simpler types (`string` and `number`, rather than `string|number`).
- It lets you declare the variables `const` rather than `let`. This makes them easier for people and the type checker to reason about.

Try to avoid type-changing variables. If you can use different names for different concepts, it will make your code clearer both to human readers and to the type checker.

This is not to be confused with "shadowed" variables as in this example:

```
const id = "12-34-56";
fetchProduct(id);

{
  const id = 123456;  // OK
  fetchProductBySerialNumber(id);  // OK
}
```

While these two `id`s share a name, they are actually two distinct variables with no relationship to one another. It's fine for them to have different types. While TypeScript is not confused by this, your human readers might be. In general it's better to use different names for different concepts. Many teams choose to disallow this sort of shadowing via linter rules.

This item focused on scalar values, but similar considerations apply to objects. For more on that, see Item 23.

Things to Remember

- While a variable's value can change, its type generally does not.

- To avoid confusion, both for human readers and for the type checker, avoid reusing variables for differently typed values.

Item 21: Understand Type Widening

As Item 7 explained, at runtime every variable has a single value. But at static analysis time, when TypeScript is checking your code, a variable has a set of *possible* values, namely, its type. When you initialize a variable with a constant but don't provide a type, the type checker needs to decide on one. In other words, it needs to decide on a set of possible values from the single value that you specified. In TypeScript, this process is known as *widening*. Understanding it will help you make sense of errors and make more effective use of type annotations.

Suppose you're writing a library to work with vectors. You write out a type for a 3D vector and a function to get the value of any of its components:

```
interface Vector3 { x: number; y: number; z: number; }
function getComponent(vector: Vector3, axis: 'x' | 'y' | 'z') {
  return vector[axis];
}
```

But when you try to use it, TypeScript flags an error:

```
let x = 'x';
let vec = {x: 10, y: 20, z: 30};
getComponent(vec, x);
               // ~ Argument of type 'string' is not assignable to
               //   parameter of type '"x" | "y" | "z"'
```

This code runs fine, so why the error?

The issue is that x's type is inferred as `string`, whereas the `getComponent` function expected a more specific type for its second argument. This is widening at work, and here it has led to an error.

This process is ambiguous in the sense that there are many possible types for any given value. In this statement, for example:

```
const mixed = ['x', 1];
```

what should the type of `mixed` be? Here are a few possibilities:

- `('x' | 1)[]`
- `['x', 1]`
- `[string, number]`
- `readonly [string, number]`
- `(string|number)[]`

- `readonly (string|number)[]`
- `[any, any]`
- `any[]`

Without more context, TypeScript has no way to know which one is "right." It has to guess at your intent. (In this case, it guesses `(string|number)[]`.) And smart as it is, TypeScript can't read your mind. It won't get this right 100% of the time. The result is inadvertent errors like the one we just looked at.

In the initial example, the type of `x` is inferred as `string` because TypeScript chooses to allow code like this:

```
let x = 'x';
x = 'a';
x = 'Four score and seven years ago...';
```

But it would also be valid JavaScript to write:

```
let x = 'x';
x = /x|y|z/;
x = ['x', 'y', 'z'];
```

In inferring the type of `x` as `string`, TypeScript attempts to strike a balance between specificity and flexibility. The general rule is that a variable's type shouldn't change after it's declared (Item 20), so `string` makes more sense than `string|RegExp` or `string|string[]` or `any`.

TypeScript gives you a few ways to control the process of widening. One is `const`. If you declare a variable with `const` instead of `let`, it gets a narrower type. In fact, using `const` fixes the error in our original example:

```
const x = 'x';  // type is "x"
let vec = {x: 10, y: 20, z: 30};
getComponent(vec, x);  // OK
```

Because `x` cannot be reassigned, TypeScript is able to infer a narrower type without risk of inadvertently flagging errors on subsequent assignments. And because the string literal type `"x"` is assignable to `"x"|"y"|"z"`, the code passes the type checker.

`const` isn't a panacea, however. For objects and arrays, there is still ambiguity. The `mixed` example here illustrates the issue for arrays: should TypeScript infer a tuple type? What type should it infer for the elements? Similar issues arise with objects. This code is fine in JavaScript:

```
const v = {
  x: 1,
};
v.x = 3;
v.x = '3';
```

```
v.y = 4;
v.name = 'Pythagoras';
```

The type of v could be inferred anywhere along the spectrum of specificity. At the specific end is {readonly x: 1}. More general is {x: number}. More general still would be {[key: string]: number} or object. In the case of objects, TypeScript's widening algorithm treats each element as though it were assigned with let. So the type of v comes out as {x: number}. This lets you reassign v.x to a different number, but not to a string. And it prevents you from adding other properties. (This is a good reason to build objects all at once, as explained in Item 23.)

So the last three statements are errors:

```
 const v = {
   x: 1,
 };
 v.x = 3;  // OK
 v.x = '3';
// ~ Type '"3"' is not assignable to type 'number'
 v.y = 4;
// ~ Property 'y' does not exist on type '{ x: number; }'
 v.name = 'Pythagoras';
// ~~~~ Property 'name' does not exist on type '{ x: number; }'
```

Again, TypeScript is trying to strike a balance between specificity and flexibility. It needs to infer a specific enough type to catch errors, but not so specific that it creates false positives. It does this by inferring a type of number for a property initialized to a value like 1.

If you know better, there are a few ways to override TypeScript's default behavior. One is to supply an explicit type annotation:

```
const v: {x: 1|3|5} = {
  x: 1,
};  // Type is { x: 1 | 3 | 5; }
```

Another is to provide additional context to the type checker (e.g., by passing the value as the parameter of a function). For much more on the role of context in type inference, see Item 26.

A third way is with a const assertion. This is not to be confused with let and const, which introduce symbols in value space. This is a purely type-level construct. Look at the different inferred types for these variables:

```
const v1 = {
  x: 1,
  y: 2,
};  // Type is { x: number; y: number; }

const v2 = {
  x: 1 as const,
```

```
  y: 2,
};  // Type is { x: 1; y: number; }

const v3 = {
  x: 1,
  y: 2,
} as const;  // Type is { readonly x: 1; readonly y: 2; }
```

When you write `as const` after a value, TypeScript will infer the narrowest possible type for it. There is *no* widening. For true constants, this is typically what you want. You can also use `as const` with arrays to infer a tuple type:

```
const a1 = [1, 2, 3];  // Type is number[]
const a2 = [1, 2, 3] as const;  // Type is readonly [1, 2, 3]
```

If you're getting incorrect errors that you think are due to widening, consider adding some explicit type annotations or `const` assertions. Inspecting types in your editor is the key to building an intuition for this (see Item 6).

Things to Remember

- Understand how TypeScript infers a type from a constant by widening it.
- Familiarize yourself with the ways you can affect this behavior: `const`, type annotations, context, and `as const`.

Item 22: Understand Type Narrowing

The opposite of widening is narrowing. This is the process by which TypeScript goes from a broad type to a narrower one. Perhaps the most common example of this is null checking:

```
const el = document.getElementById('foo'); // Type is HTMLElement | null
if (el) {
  el // Type is HTMLElement
  el.innerHTML = 'Party Time'.blink();
} else {
  el // Type is null
  alert('No element #foo');
}
```

If `el` is `null`, then the code in the first branch won't execute. So TypeScript is able to exclude `null` from the type union within this block, resulting in a narrower type which is much easier to work with. The type checker is generally quite good at narrowing types in conditionals like these, though it can occasionally be thwarted by aliasing (Item 24).

You can also narrow a variable's type for the rest of a block by throwing or returning from a branch. For example:

```
const el = document.getElementById('foo'); // Type is HTMLElement | null
if (!el) throw new Error('Unable to find #foo');
el; // Now type is HTMLElement
el.innerHTML = 'Party Time'.blink();
```

There are many ways that you can narrow a type. Using `instanceof` works:

```
function contains(text: string, search: string|RegExp) {
  if (search instanceof RegExp) {
    search  // Type is RegExp
    return !!search.exec(text);
  }
  search  // Type is string
  return text.includes(search);
}
```

So does a property check:

```
interface A { a: number }
interface B { b: number }
function pickAB(ab: A | B) {
  if ('a' in ab) {
    ab // Type is A
  } else {
    ab // Type is B
  }
  ab // Type is A | B
}
```

Some built-in functions such as `Array.isArray` are able to narrow types:

```
function contains(text: string, terms: string|string[]) {
  const termList = Array.isArray(terms) ? terms : [terms];
  termList // Type is string[]
  // ...
}
```

TypeScript is generally quite good at tracking types through conditionals. Think twice before adding an assertion—it might be onto something that you're not! For example, this is the wrong way to exclude `null` from a union type:

```
const el = document.getElementById('foo'); // type is HTMLElement | null
if (typeof el === 'object') {
  el;  // Type is HTMLElement | null
}
```

Because `typeof null` is `"object"` in JavaScript, you have not, in fact, excluded `null` with this check! Similar surprises can come from falsy primitive values:

```
function foo(x?: number|string|null) {
  if (!x) {
    x;  // Type is string | number | null | undefined
  }
}
```

Because the empty string and `0` are both falsy, `x` could still be a `string` or `number` in that branch. TypeScript is right!

Another common way to help the type checker narrow your types is by putting an explicit "tag" on them:

```
interface UploadEvent { type: 'upload'; filename: string; contents: string }
interface DownloadEvent { type: 'download'; filename: string; }
type AppEvent = UploadEvent | DownloadEvent;

function handleEvent(e: AppEvent) {
  switch (e.type) {
    case 'download':
      e  // Type is DownloadEvent
      break;
    case 'upload':
      e;  // Type is UploadEvent
      break;
  }
}
```

This pattern is known as a "tagged union" or "discriminated union," and it is ubiquitous in TypeScript.

If TypeScript isn't able to figure out a type, you can even introduce a custom function to help it out:

```
function isInputElement(el: HTMLElement): el is HTMLInputElement {
  return 'value' in el;
}

function getElementContent(el: HTMLElement) {
  if (isInputElement(el)) {
    el; // Type is HTMLInputElement
    return el.value;
  }
  el; // Type is HTMLElement
  return el.textContent;
}
```

This is known as a "user-defined type guard." The `el is HTMLInputElement` as a return type tells the type checker that it can narrow the type of the parameter if the function returns true.

Some functions are able to use type guards to perform type narrowing across arrays or objects. If you do some lookups in an array, for instance, you may wind up with an array of nullable types:

```
const jackson5 = ['Jackie', 'Tito', 'Jermaine', 'Marlon', 'Michael'];
const members = ['Janet', 'Michael'].map(
  who => jackson5.find(n => n === who)
);  // Type is (string | undefined)[]
```

If you filter out the `undefined` values using `filter`, TypeScript isn't able to follow along:

```
const members = ['Janet', 'Michael'].map(
  who => jackson5.find(n => n === who)
).filter(who => who !== undefined);  // Type is (string | undefined)[]
```

But if you use a type guard, it can:

```
function isDefined<T>(x: T | undefined): x is T {
  return x !== undefined;
}
const members = ['Janet', 'Michael'].map(
  who => jackson5.find(n => n === who)
).filter(isDefined);  // Type is string[]
```

As always, inspecting types in your editor is key to building an intuition for how narrowing works.

Understanding how types in TypeScript narrow will help you build an intuition for how type inference works, make sense of errors, and generally have a more productive relationship with the type checker.

Things to Remember

- Understand how TypeScript narrows types based on conditionals and other types of control flow.
- Use tagged/discriminated unions and user-defined type guards to help the process of narrowing.

Item 23: Create Objects All at Once

As Item 20 explained, while a variable's value may change, its type in TypeScript generally does not. This makes some JavaScript patterns easier to model in TypeScript than others. In particular, it means that you should prefer creating objects all at once, rather than piece by piece.

Here's one way to create an object representing a two-dimensional point in JavaScript:

```
const pt = {};
pt.x = 3;
pt.y = 4;
```

In TypeScript, this will produce errors on each assignment:

```
const pt = {};
pt.x = 3;
// ~ Property 'x' does not exist on type '{}'
pt.y = 4;
// ~ Property 'y' does not exist on type '{}'
```

This is because the type of `pt` on the first line is inferred based on its value `{}`, and you may only assign to known properties.

You get the opposite problem if you define a `Point` interface:

```
interface Point { x: number; y: number; }
const pt: Point = {};
   // ~~ Type '{}' is missing the following properties from type 'Point': x, y
pt.x = 3;
pt.y = 4;
```

The solution is to define the object all at once:

```
const pt = {
  x: 3,
  y: 4,
};  // OK
```

If you must build the object piecemeal, you may use a type assertion (`as`) to silence the type checker:

```
const pt = {} as Point;
pt.x = 3;
pt.y = 4;  // OK
```

But the better way is by building the object all at once and using a declaration (see Item 9):

```
const pt: Point = {
  x: 3,
  y: 4,
};
```

If you need to build a larger object from smaller ones, avoid doing it in multiple steps:

```
const pt = {x: 3, y: 4};
const id = {name: 'Pythagoras'};
const namedPoint = {};
Object.assign(namedPoint, pt, id);
namedPoint.name;
        // ~~~~ Property 'name' does not exist on type '{}'
```

You can build the larger object all at once instead using the *object spread operator*, `...`:

```
const namedPoint = {...pt, ...id};
namedPoint.name;  // OK, type is string
```

You can also use the object spread operator to build up objects field by field in a type-safe way. The key is to use a new variable on every update so that each gets a new type:

```
const pt0 = {};
const pt1 = {...pt0, x: 3};
const pt: Point = {...pt1, y: 4};  // OK
```

While this is a roundabout way to build up such a simple object, it can be a useful technique for adding properties to an object and allowing TypeScript to infer a new type.

To conditionally add a property in a type-safe way, you can use object spread with `null` or `{}`, which add no properties:

```
declare let hasMiddle: boolean;
const firstLast = {first: 'Harry', last: 'Truman'};
const president = {...firstLast, ...(hasMiddle ? {middle: 'S'} : {})};
```

If you mouse over `president` in your editor, you'll see that its type is inferred as a union:

```
const president: {
    middle: string;
    first: string;
    last: string;
} | {
    first: string;
    last: string;
}
```

This may come as a surprise if you wanted `middle` to be an optional field. You can't read `middle` off this type, for example:

```
president.middle
       // ~~~~~~ Property 'middle' does not exist on type
       //        '{ first: string; last: string; }'
```

If you're conditionally adding multiple properties, the union does more accurately represent the set of possible values (Item 32). But an optional field would be easier to work with. You can get one with a helper:

```
function addOptional<T extends object, U extends object>(
  a: T, b: U | null
): T & Partial<U> {
  return {...a, ...b};
}

const president = addOptional(firstLast, hasMiddle ? {middle: 'S'} : null);
president.middle  // OK, type is string | undefined
```

Sometimes you want to build an object or array by transforming another one. In this case the equivalent of "building objects all at once" is using built-in functional constructs or utility libraries like Lodash rather than loops. See Item 27 for more on this.

Things to Remember

- Prefer to build objects all at once rather than piecemeal. Use object spread (`{...a, ...b}`) to add properties in a type-safe way.
- Know how to conditionally add properties to an object.

Item 24: Be Consistent in Your Use of Aliases

When you introduce a new name for a value:

```
const borough = {name: 'Brooklyn', location: [40.688, -73.979]};
const loc = borough.location;
```

you have created an *alias*. Changes to properties on the alias will be visible on the original value as well:

```
> loc[0] = 0;
> borough.location
[0, -73.979]
```

Aliases are the bane of compiler writers in all languages because they make control flow analysis difficult. If you're deliberate in your use of aliases, TypeScript will be able to understand your code better and help you find more real errors.

Suppose you have a data structure that represents a polygon:

```
interface Coordinate {
  x: number;
  y: number;
}

interface BoundingBox {
  x: [number, number];
  y: [number, number];
}

interface Polygon {
  exterior: Coordinate[];
  holes: Coordinate[][];
  bbox?: BoundingBox;
}
```

The geometry of the polygon is specified by the `exterior` and `holes` properties. The `bbox` property is an optimization that may or may not be present. You can use it to speed up a point-in-polygon check:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  if (polygon.bbox) {
    if (pt.x < polygon.bbox.x[0] || pt.x > polygon.bbox.x[1] ||
        pt.y < polygon.bbox.y[1] || pt.y > polygon.bbox.y[1]) {
      return false;
    }
  }

  // ... more complex check
}
```

This code works (and type checks) but is a bit repetitive: `polygon.bbox` appears five times in three lines! Here's an attempt to factor out an intermediate variable to reduce duplication:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const box = polygon.bbox;
  if (polygon.bbox) {
    if (pt.x < box.x[0] || pt.x > box.x[1] ||
        //     ~~~                ~~~  Object is possibly 'undefined'
        pt.y < box.y[1] || pt.y > box.y[1]) {
        //     ~~~                ~~~  Object is possibly 'undefined'
      return false;
    }
  }
  // ...
}
```

(I'm assuming you've enabled `strictNullChecks`.)

This code still works, so why the error? By factoring out the `box` variable, you've created an alias for `polygon.bbox`, and this has thwarted the control flow analysis that quietly worked in the first example.

You can inspect the types of `box` and `polygon.bbox` to see what's happening:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  polygon.bbox  // Type is BoundingBox | undefined
  const box = polygon.bbox;
  box  // Type is BoundingBox | undefined
  if (polygon.bbox) {
    polygon.bbox  // Type is BoundingBox
    box  // Type is BoundingBox | undefined
  }
}
```

The property check refines the type of `polygon.bbox` but not of `box` and hence the errors. This leads us to the golden rule of aliasing: *if you introduce an alias, use it consistently*.

Using `box` in the property check fixes the error:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const box = polygon.bbox;
  if (box) {
    if (pt.x < box.x[0] || pt.x > box.x[1] ||
        pt.y < box.y[1] || pt.y > box.y[1]) {  // OK
      return false;
    }
  }
  // ...
}
```

The type checker is happy now, but there's an issue for human readers. We're using two names for the same thing: box and bbox. This is a distinction without a difference (Item 36).

Object destructuring syntax rewards consistent naming with a more compact syntax. You can even use it on arrays and nested structures:

```
function isPointInPolygon(polygon: Polygon, pt: Coordinate) {
  const {bbox} = polygon;
  if (bbox) {
    const {x, y} = bbox;
    if (pt.x < x[0] || pt.x > x[1] ||
        pt.y < x[0] || pt.y > y[1]) {
      return false;
    }
  }
  // ...
}
```

A few other points:

- This code would have required more property checks if the x and y properties had been optional, rather than the whole bbox property. We benefited from following the advice of Item 31, which discusses the importance of pushing null values to the perimeter of your types.
- An optional property was appropriate for bbox but would not have been appropriate for holes. If holes was optional, then it would be possible for it to be either missing or an empty array ([]). This would be a distinction without a difference. An empty array is a fine way to indicate "no holes."

In your interactions with the type checker, don't forget that aliasing can introduce confusion at runtime, too:

```
const {bbox} = polygon;
if (!bbox) {
  calculatePolygonBbox(polygon);  // Fills in polygon.bbox
  // Now polygon.bbox and bbox refer to different values!
}
```

TypeScript's control flow analysis tends to be quite good for local variables. But for properties you should be on guard:

```
function fn(p: Polygon) { /* ... */ }

polygon.bbox  // Type is BoundingBox | undefined
if (polygon.bbox) {
  polygon.bbox  // Type is BoundingBox
  fn(polygon);
  polygon.bbox  // Type is still BoundingBox
}
```

The call to `fn(polygon)` could very well un-set `polygon.bbox`, so it would be safer for the type to revert to `BoundingBox | undefined`. But this would get frustrating: you'd have to repeat your property checks every time you called a function. So TypeScript makes the pragmatic choice to assume the function does not invalidate its type refinements. But it *could*. If you'd factored out a local `bbox` variable instead of using `polygon.bbox`, the type of `bbox` would remain accurate, but it might no longer be the same value as `polygon.box`.

Things to Remember

- Aliasing can prevent TypeScript from narrowing types. If you create an alias for a variable, use it consistently.
- Use destructuring syntax to encourage consistent naming.
- Be aware of how function calls can invalidate type refinements on properties. Trust refinements on local variables more than on properties.

Item 25: Use async Functions Instead of Callbacks for Asynchronous Code

Classic JavaScript modeled asynchronous behavior using callbacks. This leads to the infamous "pyramid of doom":

```
fetchURL(url1, function(response1) {
  fetchURL(url2, function(response2) {
    fetchURL(url3, function(response3) {
      // ...
      console.log(1);
    });
    console.log(2);
  });
  console.log(3);
});
console.log(4);
```

```
// Logs:
// 4
// 3
// 2
// 1
```

As you can see from the logs, the execution order is the opposite of the code order. This makes callback code hard to read. It gets even more confusing if you want to run the requests in parallel or bail when an error occurs.

ES2015 introduced the concept of a Promise to break the pyramid of doom. A Promise represents something that will be available in the future (they're also sometimes called "futures"). Here's the same code using Promises:

```
const page1Promise = fetch(url1);
page1Promise.then(response1 => {
  return fetch(url2);
}).then(response2 => {
  return fetch(url3);
}).then(response3 => {
  // ...
}).catch(error => {
  // ...
});
```

Now there's less nesting, and the execution order more directly matches the code order. It's also easier to consolidate error handling and use higher-order tools like `Promise.all`.

ES2017 introduced the `async` and `await` keywords to make things even simpler:

```
async function fetchPages() {
  const response1 = await fetch(url1);
  const response2 = await fetch(url2);
  const response3 = await fetch(url3);
  // ...
}
```

The `await` keyword pauses execution of the `fetchPages` function until each Promise resolves. Within an `async` function, `await`ing a Promise that throws an exception. This lets you use the usual try/catch machinery:

```
async function fetchPages() {
  try {
    const response1 = await fetch(url1);
    const response2 = await fetch(url2);
    const response3 = await fetch(url3);
    // ...
  } catch (e) {
    // ...
  }
}
```

When you target ES5 or earlier, the TypeScript compiler will perform some elaborate transformations to make async and await work. In other words, whatever your runtime, with TypeScript you can use async/await.

There are a few good reasons to prefer Promises or async/await to callbacks:

- Promises are easier to compose than callbacks.
- Types are able to flow through Promises more easily than callbacks.

If you want to fetch the pages in parallel, for example, you can compose Promises with Promise.all:

```
async function fetchPages() {
  const [response1, response2, response3] = await Promise.all([
    fetch(url1), fetch(url2), fetch(url3)
  ]);
  // ...
}
```

Using destructuring assignment with await is particularly nice in this context.

TypeScript is able to infer the types of each of the three response variables as Response. The equivalent code to do the requests in parallel with callbacks requires more machinery and a type annotation:

```
function fetchPagesCB() {
  let numDone = 0;
  const responses: string[] = [];
  const done = () => {
    const [response1, response2, response3] = responses;
    // ...
  };
  const urls = [url1, url2, url3];
  urls.forEach((url, i) => {
    fetchURL(url, r => {
      responses[i] = url;
      numDone++;
      if (numDone === urls.length) done();
    });
  });
}
```

Extending this to include error handling or to be as generic as Promise.all is challenging.

Type inference also works well with Promise.race, which resolves when the first of its input Promises resolves. You can use this to add timeouts to Promises in a general way:

```
function timeout(millis: number): Promise<never> {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject('timeout'), millis);
  });
}

async function fetchWithTimeout(url: string, ms: number) {
  return Promise.race([fetch(url), timeout(ms)]);
}
```

The return type of `fetchWithTimeout` is inferred as `Promise<Response>`, no type annotations required. It's interesting to dig into why this works: the return type of `Promise.race` is the union of the types of its inputs, in this case `Promise<Response | never>`. But taking a union with `never` (the empty set) is a no-op, so this gets simplified to `Promise<Response>`. When you work with Promises, all of TypeScript's type inference machinery works to get you the right types.

There are some times when you need to use raw Promises, notably when you are wrapping a callback API like `setTimeout`. But if you have a choice, you should generally prefer `async`/`await` to raw Promises for two reasons:

- It typically produces more concise and straightforward code.
- It enforces that `async` functions always return Promises.

An `async` function always returns a `Promise`, even if it doesn't involve `await`ing anything. TypeScript can help you build an intuition for this:

```
// function getNumber(): Promise<number>
async function getNumber() {
  return 42;
}
```

You can also create `async` arrow functions:

```
const getNumber = async () => 42;  // Type is () => Promise<number>
```

The raw Promise equivalent is:

```
const getNumber = () => Promise.resolve(42);  // Type is () => Promise<number>
```

While it may seem odd to return a Promise for an immediately available value, this actually helps enforce an important rule: a function should either always be run synchronously or always be run asynchronously. It should never mix the two. For example, what if you want to add a cache to the `fetchURL` function? Here's an attempt:

```
// Don't do this!
const _cache: {[url: string]: string} = {};
function fetchWithCache(url: string, callback: (text: string) => void) {
  if (url in _cache) {
    callback(_cache[url]);
  } else {
    fetchURL(url, text => {
      _cache[url] = text;
      callback(text);
    });
  }
}
```

While this may seem like an optimization, the function is now extremely difficult for a client to use:

```
let requestStatus: 'loading' | 'success' | 'error';
function getUser(userId: string) {
  fetchWithCache(`/user/${userId}`, profile => {
    requestStatus = 'success';
  });
  requestStatus = 'loading';
}
```

What will the value of `requestStatus` be after calling `getUser`? It depends entirely on whether the profile is cached. If it's not, `requestStatus` will be set to "success." If it is, it'll get set to "success" and then set back to "loading." Oops!

Using `async` for both functions enforces consistent behavior:

```
const _cache: {[url: string]: string} = {};
async function fetchWithCache(url: string) {
  if (url in _cache) {
    return _cache[url];
  }
  const response = await fetch(url);
  const text = await response.text();
  _cache[url] = text;
  return text;
}

let requestStatus: 'loading' | 'success' | 'error';
async function getUser(userId: string) {
  requestStatus = 'loading';
  const profile = await fetchWithCache(`/user/${userId}`);
  requestStatus = 'success';
}
```

Now it's completely transparent that `requestStatus` will end in "success." It's easy to accidentally produce half-synchronous code with callbacks or raw Promises, but difficult with `async`.

Note that if you return a Promise from an `async` function, it will not get wrapped in another Promise: the return type will be `Promise<T>` rather than `Promise<Promise<T>>`. Again, TypeScript will help you build an intuition for this:

```
// Function getJSON(url: string): Promise<any>
async function getJSON(url: string) {
  const response = await fetch(url);
  const jsonPromise = response.json();  // Type is Promise<any>
  return jsonPromise;
}
```

Things to Remember

- Prefer Promises to callbacks for better composability and type flow.
- Prefer `async` and `await` to raw Promises when possible. They produce more concise, straightforward code and eliminate whole classes of errors.
- If a function returns a Promise, declare it `async`.

Item 26: Understand How Context Is Used in Type Inference

TypeScript doesn't just infer types based on values. It also considers the context in which the value occurs. This usually works well but can sometimes lead to surprises. Understanding how context is used in type inference will help you identify and work around these surprises when they do occur.

In JavaScript you can factor an expression out into a constant without changing the behavior of your code (so long as you don't alter execution order). In other words, these two statements are equivalent:

```
// Inline form
setLanguage('JavaScript');

// Reference form
let language = 'JavaScript';
setLanguage(language);
```

In TypeScript, this refactor still works:

```
function setLanguage(language: string) { /* ... */ }

setLanguage('JavaScript');  // OK

let language = 'JavaScript';
setLanguage(language);  // OK
```

Now suppose you take to heart the advice of Item 33 and replace the string type with a more precise union of string literal types:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
function setLanguage(language: Language) { /* ... */ }

setLanguage('JavaScript');  // OK

let language = 'JavaScript';
setLanguage(language);
```

```
         // ~~~~~~~~ Argument of type 'string' is not assignable
         //          to parameter of type 'Language'
```

What went wrong? With the inline form, TypeScript knows from the function declaration that the parameter is supposed to be of type `Language`. The string literal `'JavaScript'` is assignable to this type, so this is OK. But when you factor out a variable, TypeScript must infer its type at the time of assignment. In this case it infers `string`, which is not assignable to `Language`. Hence the error.

(Some languages are able to infer types for variables based on their eventual usage. But this can also be confusing. Anders Hejlsberg, the creator of TypeScript, refers to it as "spooky action at a distance." By and large, TypeScript determines the type of a variable when it is first introduced. For a notable exception to this rule, see Item 41.)

There are two good ways to solve this problem. One is to constrain the possible values of `language` with a type declaration:

```
let language: Language = 'JavaScript';
setLanguage(language);  // OK
```

This also has the benefit of flagging an error if there's a typo in the language—for example `'Typescript'` (it should be a capital "S").

The other solution is to make the variable constant:

```
const language = 'JavaScript';
setLanguage(language);  // OK
```

By using `const`, we've told the type checker that this variable cannot change. So TypeScript can infer a more precise type for `language`, the string literal type `"JavaScript"`. This is assignable to `Language` so the code type checks. Of course, if you do need to reassign `language`, then you'll need to use the type declaration. (For more on this, see Item 21.)

The fundamental issue here is that we've separated the value from the context in which it's used. Sometimes this is OK, but often it is not. The rest of this item walks through a few cases where this loss of context can cause errors and shows you how to fix them.

Tuple Types

In addition to string literal types, problems can come up with tuple types. Suppose you're working with a map visualization that lets you programmatically pan the map:

```
// Parameter is a (latitude, longitude) pair.
function panTo(where: [number, number]) { /* ... */ }

panTo([10, 20]);  // OK

const loc = [10, 20];
```

```
panTo(loc);
//    ~~~ Argument of type 'number[]' is not assignable to
//        parameter of type '[number, number]'
```

As before, you've separated a value from its context. In the first instance `[10, 20]` is assignable to the tuple type `[number, number]`. In the second, TypeScript infers the type of `loc` as `number[]` (i.e., an array of numbers of unknown length). This is not assignable to the tuple type, since many arrays have the wrong number of elements.

So how can you fix this error without resorting to `any`? You've already declared it `const`, so that won't help. But you can still provide a type declaration to let TypeScript know precisely what you mean:

```
const loc: [number, number] = [10, 20];
panTo(loc);  // OK
```

Another way is to provide a "const context." This tells TypeScript that you intend the value to be deeply constant, rather than the shallow constant that `const` gives:

```
const loc = [10, 20] as const;
panTo(loc);
   // ~~~ Type 'readonly [10, 20]' is 'readonly'
   //     and cannot be assigned to the mutable type '[number, number]'
```

If you hover over `loc` in your editor, you'll see that its type is now inferred as `readonly [10, 20]`, rather than `number[]`. Unfortunately this is *too* precise! The type signature of `panTo` makes no promises that it won't modify the contents of its `where` parameter. Since the `loc` parameter has a `readonly` type, this won't do. The best solution here is to add a `readonly` annotation to the `panTo` function:

```
function panTo(where: readonly [number, number]) { /* ... */ }
const loc = [10, 20] as const;
panTo(loc);  // OK
```

If the type signature is outside your control, then you'll need to use an annotation.

`const` contexts can neatly solve issues around losing context in inference, but they do have an unfortunate downside: if you make a mistake in the definition (say you add a third element to the tuple) then the error will be flagged at the call site, not at the definition. This may be confusing, especially if the error occurs in a deeply nested object:

```
const loc = [10, 20, 30] as const;  // error is really here.
panTo(loc);
//    ~~~ Argument of type 'readonly [10, 20, 30]' is not assignable to
//        parameter of type 'readonly [number, number]'
//          Types of property 'length' are incompatible
//            Type '3' is not assignable to type '2'
```

Objects

The problem of separating a value from its context also comes up when you factor out a constant from a larger object that contains some string literals or tuples. For example:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
interface GovernedLanguage {
  language: Language;
  organization: string;
}

function complain(language: GovernedLanguage) { /* ... */ }

complain({ language: 'TypeScript', organization: 'Microsoft' });  // OK

const ts = {
  language: 'TypeScript',
  organization: 'Microsoft',
};
complain(ts);
//       ~~ Argument of type '{ language: string; organization: string; }'
//            is not assignable to parameter of type 'GovernedLanguage'
//          Types of property 'language' are incompatible
//            Type 'string' is not assignable to type 'Language'
```

In the `ts` object, the type of `language` is inferred as `string`. As before, the solution is to add a type declaration (`const ts: GovernedLanguage = ...`) or use a const assertion (`as const`).

Callbacks

When you pass a callback to another function, TypeScript uses context to infer the parameter types of the callback:

```
function callWithRandomNumbers(fn: (n1: number, n2: number) => void) {
  fn(Math.random(), Math.random());
}

callWithRandomNumbers((a, b) => {
  a;  // Type is number
  b;  // Type is number
  console.log(a + b);
});
```

The types of `a` and `b` are inferred as `number` because of the type declaration for `callWithRandom`. If you factor the callback out into a constant, you lose that context and get `noImplicitAny` errors:

```
const fn = (a, b) => {
         // ~    Parameter 'a' implicitly has an 'any' type
```

```
        //    ~ Parameter 'b' implicitly has an 'any' type
  console.log(a + b);
}
callWithRandomNumbers(fn);
```

The solution is either to add type annotations to the parameters:

```
const fn = (a: number, b: number) => {
  console.log(a + b);
}
callWithRandomNumbers(fn);
```

or to apply a type declaration to the entire function expression if one is available. See Item 12.

Things to Remember

- Be aware of how context is used in type inference.
- If factoring out a variable introduces a type error, consider adding a type declaration.
- If the variable is truly a constant, use a const assertion (`as const`). But be aware that this may result in errors surfacing at use, rather than definition.

Item 27: Use Functional Constructs and Libraries to Help Types Flow

JavaScript has never included the sort of standard library you find in Python, C, or Java. Over the years many libraries have tried to fill the gap. jQuery provided helpers not just for interacting with the DOM but also for iterating and mapping over objects and arrays. Underscore focused more on providing general utility functions, and Lodash built on this effort. Today libraries like Ramda continue to bring ideas from functional programming into the JavaScript world.

Some features from these libraries, such as `map`, `flatMap`, `filter`, and `reduce`, have made it into the JavaScript language itself. While these constructs (and the other ones provided by Lodash) are helpful in JavaScript and often preferable to a hand-rolled loop, this advantage tends to get even more lopsided when you add TypeScript to the mix. This is because their type declarations ensure that types flow through these constructs. With hand-rolled loops, you're responsible for the types yourself.

For example, consider parsing some CSV data. You could do it in plain JavaScript in a somewhat imperative style:

```
const csvData = "...";
const rawRows = csvData.split('\n');
const headers = rawRows[0].split(',');
const rows = rawRows.slice(1).map(rowStr => {
  const row = {};
  rowStr.split(',').forEach((val, j) => {
    row[headers[j]] = val;
  });
  return row;
});
```

More functionally minded JavaScripters might prefer to build the row objects with `reduce`:

```
const rows = rawRows.slice(1)
    .map(rowStr => rowStr.split(',').reduce(
        (row, val, i) => (row[headers[i]] = val, row),
        {}));
```

This version saves three lines (almost 20 non-whitespace characters!) but may be more cryptic depending on your sensibilities. Lodash's `zipObject` function, which forms an object by "zipping" up a keys and values array, can tighten it even further:

```
import _ from 'lodash';
const rows = rawRows.slice(1)
    .map(rowStr => _.zipObject(headers, rowStr.split(',')));
```

I find this the clearest of all. But is it worth the cost of adding a dependency on a third-party library to your project? If you're not using a bundler and the overhead of doing this is significant, then the answer may be "no."

When you add TypeScript to the mix, it starts to tip the balance more strongly in favor of the Lodash solution.

Both vanilla JS versions of the CSV parser produce the same error in TypeScript:

```
const rowsA = rawRows.slice(1).map(rowStr => {
  const row = {};
  rowStr.split(',').forEach((val, j) => {
    row[headers[j]] = val;
 // ~~~~~~~~~~~~~~~ No index signature with a parameter of
 //                 type 'string' was found on type '{}'
  });
  return row;
});
const rowsB = rawRows.slice(1)
  .map(rowStr => rowStr.split(',').reduce(
      (row, val, i) => (row[headers[i]] = val, row),
                     // ~~~~~~~~~~~~~~~ No index signature with a parameter of
                     //                 type 'string' was found on type '{}'
      {}));
```

The solution in each case is to provide a type annotation for `{}`, either `{[column: string]: string}` or `Record<string, string>`.

The Lodash version, on the other hand, passes the type checker without modification:

```
const rows = rawRows.slice(1)
    .map(rowStr => _.zipObject(headers, rowStr.split(',')));
    // Type is _.Dictionary<string>[]
```

`Dictionary` is a Lodash type alias. `Dictionary<string>` is the same as `{[key: string]: string}` or `Record<string, string>`. The important thing here is that the type of `rows` is exactly correct, no type annotations needed.

These advantages get more pronounced as your data munging gets more elaborate. For example, suppose you have a list of the rosters for all the NBA teams:

```
interface BasketballPlayer {
  name: string;
  team: string;
  salary: number;
}
declare const rosters: {[team: string]: BasketballPlayer[]};
```

To build a flat list using a loop, you might use `concat` with an array. This code runs fine but does not type check:

```
let allPlayers = [];
 // ~~~~~~~~~~ Variable 'allPlayers' implicitly has type 'any[]'
 //            in some locations where its type cannot be determined
for (const players of Object.values(rosters)) {
  allPlayers = allPlayers.concat(players);
           // ~~~~~~~~~~ Variable 'allPlayers' implicitly has an 'any[]' type
}
```

To fix the error you need to add a type annotation to `allPlayers`:

```
let allPlayers: BasketballPlayer[] = [];
for (const players of Object.values(rosters)) {
  allPlayers = allPlayers.concat(players);  // OK
}
```

But a better solution is to use `Array.prototype.flat`:

```
const allPlayers = Object.values(rosters).flat();
// OK, type is BasketballPlayer[]
```

The `flat` method flattens a multidimensional array. Its type signature is something like `T[][] => T[]`. This version is the most concise and requires no type annotations. As an added bonus you can use `const` instead of `let` to prevent future mutations to the `allPlayers` variable.

Say you want to start with `allPlayers` and make a list of the highest-paid players on each team ordered by salary.

Here's a solution without Lodash. It requires a type annotation where you don't use functional constructs:

```
const teamToPlayers: {[team: string]: BasketballPlayer[]} = {};
for (const player of allPlayers) {
  const {team} = player;
  teamToPlayers[team] = teamToPlayers[team] || [];
  teamToPlayers[team].push(player);
}

for (const players of Object.values(teamToPlayers)) {
  players.sort((a, b) => b.salary - a.salary);
}

const bestPaid = Object.values(teamToPlayers).map(players => players[0]);
bestPaid.sort((playerA, playerB) => playerB.salary - playerA.salary);
console.log(bestPaid);
```

Here's the output:

```
[
  { team: 'GSW', salary: 37457154, name: 'Stephen Curry' },
  { team: 'HOU', salary: 35654150, name: 'Chris Paul' },
  { team: 'LAL', salary: 35654150, name: 'LeBron James' },
  { team: 'OKC', salary: 35654150, name: 'Russell Westbrook' },
  { team: 'DET', salary: 32088932, name: 'Blake Griffin' },
  ...
]
```

Here's the equivalent with Lodash:

```
const bestPaid = _(allPlayers)
  .groupBy(player => player.team)
  .mapValues(players => _.maxBy(players, p => p.salary)!)
  .values()
  .sortBy(p => -p.salary)
  .value()  // Type is BasketballPlayer[]
```

In addition to being half the length, this code is clearer and requires only a single non-null assertion (the type checker doesn't know that the `players` array passed to `_.maxBy` is non-empty). It makes use of a "chain," a concept in Lodash and Underscore that lets you write a sequence of operations in a more natural order. Instead of writing:

```
_.a(_.b(_.c(v)))
```

you write:

```
_(v).a().b().c().value()
```

The `_(v)` "wraps" the value, and the `.value()` "unwraps" it.

You can inspect each function call in the chain to see the type of the wrapped value. It's always correct.

Even some of the quirkier shorthands in Lodash can be modeled accurately in TypeScript. For instance, why would you want to use `_.map` instead of the built-in

`Array.prototype.map`? One reason is that instead of passing in a callback you can pass in the name of a property. These calls all produce the same result:

```
const namesA = allPlayers.map(player => player.name)  // Type is string[]
const namesB = _.map(allPlayers, player => player.name)  // Type is string[]
const namesC = _.map(allPlayers, 'name');  // Type is string[]
```

It's a testament to the sophistication of TypeScript's type system that it can model a construct like this accurately, but it naturally falls out of the combination of string literal types and index types (see Item 14). If you're used to C++ or Java, this sort of type inference can feel quite magical!

```
const salaries = _.map(allPlayers, 'salary');  // Type is number[]
const teams = _.map(allPlayers, 'team');  // Type is string[]
const mix = _.map(allPlayers, Math.random() < 0.5 ? 'name' : 'salary');
  // Type is (string | number)[]
```

It's not a coincidence that types flow so well through built-in functional constructs and those in libraries like Lodash. By avoiding mutation and returning new values from every call, they are able to produce new types as well (Item 20). And to a large extent, the development of TypeScript has been driven by an attempt to accurately model the behavior of JavaScript libraries in the wild. Take advantage of all this work and use them!

Things to Remember

- Use built-in functional constructs and those in utility libraries like Lodash instead of hand-rolled constructs to improve type flow, increase legibility, and reduce the need for explicit type annotations.