CHAPTER 4

Type Design

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

—Fred Brooks, *The Mythical Man Month*

The language in Fred Brooks's quote is dated, but the sentiment remains true: code is difficult to understand if you can't see the data or data types on which it operates. This is one of the great advantages of a type system: by writing out types, you make them visible to readers of your code. And this makes your code understandable.

Other chapters cover the nuts and bolts of TypeScript types: using them, inferring them, and writing declarations with them. This chapter discusses the design of the types themselves. The examples in this chapter are all written with TypeScript in mind, but most of the ideas are more broadly applicable.

If you write your types well, then with any luck your flowcharts will be obvious, too.

Item 28: Prefer Types That Always Represent Valid States

If you design your types well, your code should be straightforward to write. But if you design your types poorly, no amount of cleverness or documentation will save you. Your code will be confusing and bug prone.

A key to effective type design is crafting types that can only represent a valid state. This item walks through a few examples of how this can go wrong and shows you how to fix them.

Suppose you're building a web application that lets you select a page, loads the content of that page, and then displays it. You might write the state like this:

```
interface State {
  pageText: string;
  isLoading: boolean;
  error?: string;
}
```

When you write your code to render the page, you need to consider all of these fields:

```
function renderPage(state: State) {
  if (state.error) {
    return `Error! Unable to load ${currentPage}: ${state.error}`;
  } else if (state.isLoading) {
    return `Loading ${currentPage}...`;
  }
  return `<h1>${currentPage}</h1>\n${state.pageText}`;
}
```

Is this right, though? What if `isLoading` and `error` are both set? What would that mean? Is it better to display the loading message or the error message? It's hard to say! There's not enough information available.

Or what if you're writing a `changePage` function? Here's an attempt:

```
async function changePage(state: State, newPage: string) {
  state.isLoading = true;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const text = await response.text();
    state.isLoading = false;
    state.pageText = text;
  } catch (e) {
    state.error = '' + e;
  }
}
```

There are many problems with this! Here are a few:

- We forgot to set `state.isLoading` to `false` in the error case.
- We didn't clear out `state.error`, so if the previous request failed, then you'll keep seeing that error message instead of a loading message.
- If the user changes pages again while the page is loading, who knows what will happen. They might see a new page and then an error, or the first page and not the second depending on the order in which the responses come back.

The problem is that the state includes both too little information (which request failed? which is loading?) and too much: the `State` type allows both `isLoading` and `error` to be set, even though this represents an invalid state. This makes both `render()` and `changePage()` impossible to implement well.

Here's a better way to represent the application state:

```
interface RequestPending {
  state: 'pending';
}
interface RequestError {
  state: 'error';
  error: string;
}
interface RequestSuccess {
  state: 'ok';
  pageText: string;
}
type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: {[page: string]: RequestState};
}
```

This uses a tagged union (also known as a "discriminated union") to explicitly model the different states that a network request can be in. This version of the state is three to four times longer, but it has the enormous advantage of not admitting invalid states. The current page is modeled explicitly, as is the state of every request that you issue. As a result, the `renderPage` and `changePage` functions are easy to implement:

```
function renderPage(state: State) {
  const {currentPage} = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case 'pending':
      return `Loading ${currentPage}...`;
    case 'error':
      return `Error! Unable to load ${currentPage}: ${requestState.error}`;
    case 'ok':
      return `<h1>${currentPage}</h1>\n${requestState.pageText}`;
  }
}

async function changePage(state: State, newPage: string) {
  state.requests[newPage] = {state: 'pending'};
  state.currentPage = newPage;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const pageText = await response.text();
    state.requests[newPage] = {state: 'ok', pageText};
  } catch (e) {
    state.requests[newPage] = {state: 'error', error: '' + e};
    }
  }
```

The ambiguity from the first implementation is entirely gone: it's clear what the current page is, and every request is in exactly one state. If the user changes the page after a request has been issued, that's no problem either. The old request still completes, but it doesn't affect the UI.

For a simpler but more dire example, consider the fate of Air France Flight 447, an Airbus 330 that disappeared over the Atlantic on June 1, 2009. The Airbus was a fly-by-wire aircraft, meaning that the pilots' control inputs went through a computer system before affecting the physical control surfaces of the plane. In the wake of the crash there were many questions raised about the wisdom of relying on computers to make such life-and-death decisions. Two years later when the black box recorders were recovered, they revealed many factors that led to the crash. But a key one was bad state design.

The cockpit of the Airbus 330 had a separate set of controls for the pilot and copilot. The "side sticks" controlled the angle of attack. Pulling back would send the airplane into a climb, while pushing forward would make it dive. The Airbus 330 used a system called "dual input" mode, which let the two side sticks move independently. Here's how you might model its state in TypeScript:

```
interface CockpitControls {
  /** Angle of the left side stick in degrees, 0 = neutral, + = forward */
  leftSideStick: number;
  /** Angle of the right side stick in degrees, 0 = neutral, + = forward */
  rightSideStick: number;
}
```

Suppose you were given this data structure and asked to write a `getStickSetting` function that computed the current stick setting. How would you do it?

One way would be to assume that the pilot (who sits on the left) is in control:

```
function getStickSetting(controls: CockpitControls) {
  return controls.leftSideStick;
}
```

But what if the copilot has taken control? Maybe you should use whichever stick is away from zero:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  }
  return leftSideStick;
}
```

But there's a problem with this implementation: we can only be confident returning the left setting if the right one is neutral. So you should check for that:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  } else if (rightSideStick === 0) {
    return leftSideStick;
  }
  // ???
}
```

What do you do if they're both non-zero? Hopefully they're about the same, in which case you could just average them:

```
function getStickSetting(controls: CockpitControls) {
  const {leftSideStick, rightSideStick} = controls;
  if (leftSideStick === 0) {
    return rightSideStick;
  } else if (rightSideStick === 0) {
    return leftSideStick;
  }
  if (Math.abs(leftSideStick - rightSideStick) < 5) {
    return (leftSideStick + rightSideStick) / 2;
  }
  // ???
}
```

But what if they're not? Can you throw an error? Not really: the ailerons need to be set at some angle!

On Air France 447, the copilot silently pulled back on his side stick as the plane entered a storm. It gained altitude but eventually lost speed and entered a stall, a condition in which the plane is moving too slowly to effectively generate lift. It began to drop.

To escape a stall, pilots are trained to push the controls forward to make the plane dive and regain speed. This is exactly what the pilot did. But the copilot was still silently pulling back on his side stick. And the Airbus function looked like this:

```
function getStickSetting(controls: CockpitControls) {
  return (controls.leftSideStick + controls.rightSideStick) / 2;
}
```

Even though the pilot pushed the stick fully forward, it averaged out to nothing. He had no idea why the plane wasn't diving. By the time the copilot revealed what he'd done, the plane had lost too much altitude to recover and it crashed into the ocean, killing all 228 people on board.

The point of all this is that there is no good way to implement `getStickSetting` given that input! The function has been set up to fail. In most planes the two sets of controls are mechanically connected. If the copilot pulls back, the pilot's controls will also pull back. The state of these controls is simple to express:

```
interface CockpitControls {
  /** Angle of the stick in degrees, 0 = neutral, + = forward */
  stickAngle: number;
}
```

And now, as in the Fred Brooks quote from the start of the chapter, our flowcharts are obvious. You don't need a `getStickSetting` function at all.

As you design your types, take care to think about which values you are including and which you are excluding. If you only allow values that represent valid states, your code will be easier to write and TypeScript will have an easier time checking it. This is a very general principle, and several of the other items in this chapter will cover specific manifestations of it.

Things to Remember

- Types that represent both valid and invalid states are likely to lead to confusing and error-prone code.
- Prefer types that only represent valid states. Even if they are longer or harder to express, they will save you time and pain in the end!

Item 29: Be Liberal in What You Accept and Strict in What You Produce

This idea is known as the *robustness principle* or *Postel's Law*, after Jon Postel, who wrote it in the context of TCP:

> TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

A similar rule applies to the contracts for functions. It's fine for your functions to be broad in what they accept as inputs, but they should generally be more specific in what they produce as outputs.

As an example, a 3D mapping API might provide a way to position the camera and to calculate a viewport for a bounding box:

```
declare function setCamera(camera: CameraOptions): void;
declare function viewportForBounds(bounds: LngLatBounds): CameraOptions;
```

It is convenient that the result of `viewportForBounds` can be passed directly to `setCamera` to position the camera.

Let's look at the definitions of these types:

```
interface CameraOptions {
  center?: LngLat;
  zoom?: number;
  bearing?: number;
  pitch?: number;
}
type LngLat =
  { lng: number; lat: number; } |
  { lon: number; lat: number; } |
  [number, number];
```

The fields in `CameraOptions` are all optional because you might want to set just the center or zoom without changing the bearing or pitch. The `LngLat` type also makes `setCamera` liberal in what it accepts: you can pass in a `{lng, lat}` object, a `{lon, lat}` object, or a `[lng, lat]` pair if you're confident you got the order right. These accommodations make the function easy to call.

The `viewportForBounds` function takes in another "liberal" type:

```
type LngLatBounds =
  {northeast: LngLat, southwest: LngLat} |
  [LngLat, LngLat] |
  [number, number, number, number];
```

You can specify the bounds either using named corners, a pair of lat/lngs, or a four-tuple if you're confident you got the order right. Since `LngLat` already accommodates three forms, there are no fewer than 19 possible forms for `LngLatBounds`. Liberal indeed!

Now let's write a function that adjusts the viewport to accommodate a GeoJSON Feature and stores the new viewport in the URL (for a definition of `calculateBoundingBox`, see Item 31):

```
function focusOnFeature(f: Feature) {
  const bounds = calculateBoundingBox(f);
  const camera = viewportForBounds(bounds);
  setCamera(camera);
  const {center: {lat, lng}, zoom} = camera;
               // ~~~      Property 'lat' does not exist on type ...
               //      ~~~ Property 'lng' does not exist on type ...
  zoom;  // Type is number | undefined
  window.location.search = `?v=@${lat},${lng}z${zoom}`;
}
```

Whoops! Only the `zoom` property exists, but its type is inferred as `number|undefined`, which is also problematic. The issue is that the type declaration for `viewportForBounds` indicates that it is liberal not just in what it accepts but also in what it *produces*. The only type-safe way to use the `camera` result is to introduce a code branch for each component of the union type (Item 22).

The return type with lots of optional properties and union types makes `viewportForBounds` difficult to use. Its broad parameter type is convenient, but its broad return type is not. A more convenient API would be strict in what it produces.

One way to do this is to distinguish a canonical format for coordinates. Following JavaScript's convention of distinguishing "Array" and "Array-like" (Item 16), you can draw a distinction between `LngLat` and `LngLatLike`. You can also distinguish between a fully defined `Camera` type and the partial version accepted by `setCamera`:

```
interface LngLat { lng: number; lat: number; };
type LngLatLike = LngLat | { lon: number; lat: number; } | [number, number];

interface Camera {
  center: LngLat;
  zoom: number;
  bearing: number;
  pitch: number;
}
interface CameraOptions extends Omit<Partial<Camera>, 'center'> {
  center?: LngLatLike;
}
type LngLatBounds =
  {northeast: LngLatLike, southwest: LngLatLike} |
  [LngLatLike, LngLatLike] |
  [number, number, number, number];

declare function setCamera(camera: CameraOptions): void;
declare function viewportForBounds(bounds: LngLatBounds): Camera;
```

The loose `CameraOptions` type adapts the stricter `Camera` type (Item 14).

Using `Partial<Camera>` as the parameter type in `setCamera` would not work here since you do want to allow `LngLatLike` objects for the `center` property. And you can't write "`CameraOptions extends Partial<Camera>`" since `LngLatLike` is a superset of `LngLat`, not a subset (Item 7). If this seems too complicated, you could also write the type out explicitly at the cost of some repetition:

```
interface CameraOptions {
  center?: LngLatLike;
  zoom?: number;
  bearing?: number;
  pitch?: number;
}
```

In either case, with these new type declarations the `focusOnFeature` function passes the type checker:

```
function focusOnFeature(f: Feature) {
  const bounds = calculateBoundingBox(f);
  const camera = viewportForBounds(bounds);
  setCamera(camera);
  const {center: {lat, lng}, zoom} = camera;  // OK
  zoom;  // Type is number
  window.location.search = `?v=@${lat},${lng}z${zoom}`;
}
```

This time the type of `zoom` is `number`, rather than `number|undefined`. The `viewportForBounds` function is now much easier to use. If there were any other functions that produced bounds, you would also need to introduce a canonical form and a distinction between `LngLatBounds` and `LngLatBoundsLike`.

Is allowing 19 possible forms of bounding box a good design? Perhaps not. But if you're writing type declarations for a library that does this, you need to model its behavior. Just don't have 19 return types!

Things to Remember

- Input types tend to be broader than output types. Optional properties and union types are more common in parameter types than return types.
- To reuse types between parameters and return types, introduce a canonical form (for return types) and a looser form (for parameters).

Item 30: Don't Repeat Type Information in Documentation

What's wrong with this code?

```
/**
 * Returns a string with the foreground color.
 * Takes zero or one arguments. With no arguments, returns the
 * standard foreground color. With one argument, returns the foreground color
 * for a particular page.
 */
function getForegroundColor(page?: string) {
  return page === 'login' ? {r: 127, g: 127, b: 127} : {r: 0, g: 0, b: 0};
}
```

The code and the comment disagree! Without more context it's hard to say which is right, but something is clearly amiss. As a professor of mine used to say, "when your code and your comments disagree, they're both wrong!"

Let's assume that the code represents the desired behavior. There are a few issues with this comment:

- It says that the function returns the color as a `string` when it actually returns an `{r, g, b}` object.

- It explains that the function takes zero or one arguments, which is already clear from the type signature.
- It's needlessly wordy: the comment is longer than the function declaration *and* implementation!

TypeScript's type annotation system is designed to be compact, descriptive, and readable. Its developers are language experts with decades of experience. It's almost certainly a better way to express the types of your function's inputs and outputs than your prose!

And because your type annotations are checked by the TypeScript compiler, they'll never get out of sync with the implementation. Perhaps `getForegroundColor` used to return a string but was later changed to return an object. The person who made the change might have forgotten to update the long comment.

Nothing stays in sync unless it's forced to. With type annotations, TypeScript's type checker is that force! If you put type information in annotations and not in documentation, you greatly increase your confidence that it will remain correct as the code evolves.

A better comment might look like this:

```
/** Get the foreground color for the application or a specific page. */
function getForegroundColor(page?: string): Color {
  // ...
}
```

If you want to describe a particular parameter, use an `@param` JSDoc annotation. See Item 48 for more on this.

Comments about a lack of mutation are also suspect. Don't just say that you don't modify a parameter:

```
/** Does not modify nums */
function sort(nums: number[]) { /* ... */ }
```

Instead, declare it `readonly` (Item 17) and let TypeScript enforce the contract:

```
function sort(nums: readonly number[]) { /* ... */ }
```

What's true for comments is also true for variable names. Avoid putting types in them: rather than naming a variable `ageNum`, name it `age` and make sure it's really a `number`.

An exception to this is for numbers with units. If it's not clear what the units are, you may want to include them in a variable or property name. For instance, `timeMs` is a much clearer name than just `time`, and `temperatureC` is a much clearer name than `temperature`. Item 37 describes "brands," which provide a more type-safe approach to modeling units.

Things to Remember

- Avoid repeating type information in comments and variable names. In the best case it is duplicative of type declarations, and in the worst it will lead to conflicting information.
- Consider including units in variable names if they aren't clear from the type (e.g., `timeMs` or `temperatureC`).

Item 31: Push Null Values to the Perimeter of Your Types

When you first turn on `strictNullChecks`, it may seem as though you have to add scores of if statements checking for `null` and `undefined` values throughout your code. This is often because the relationships between null and non-null values are implicit: when variable A is non-null, you know that variable B is also non-null and vice versa. These implicit relationships are confusing both for human readers of your code and for the type checker.

Values are easier to work with when they're either completely null or completely non-null, rather than a mix. You can model this by pushing the null values out to the perimeter of your structures.

Suppose you want to calculate the min and max of a list of numbers. We'll call this the "extent." Here's an attempt:

```
function extent(nums: number[]) {
  let min, max;
  for (const num of nums) {
    if (!min) {
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
    }
  }
  return [min, max];
}
```

The code type checks (without `strictNullChecks`) and has an inferred return type of `number[]`, which seems fine. But it has a bug and a design flaw:

- If the min or max is zero, it may get overridden. For example, `extent([0, 1, 2])` will return `[1, 2]` rather than `[0, 2]`.
- If the `nums` array is empty, the function will return `[undefined, undefined]`. This sort of object with several `undefined`s will be difficult for clients to work

with and is exactly the sort of type that this item discourages. We know from reading the source code that `min` and `max` will either both be `undefined` or neither, but that information isn't represented in the type system.

Turning on `strictNullChecks` makes both of these issues more apparent:

```
function extent(nums: number[]) {
  let min, max;
  for (const num of nums) {
    if (!min) {
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
                  // ~~~ Argument of type 'number | undefined' is not
                  //     assignable to parameter of type 'number'
    }
  }
  return [min, max];
}
```

The return type of `extent` is now inferred as `(number | undefined)[]`, which makes the design flaw more apparent. This is likely to manifest as a type error wherever you call `extent`:

```
const [min, max] = extent([0, 1, 2]);
const span = max - min;
          // ~~~   ~~~ Object is possibly 'undefined'
```

The error in the implementation of `extent` comes about because you've excluded `undefined` as a value for `min` but not `max`. The two are initialized together, but this information isn't present in the type system. You could make it go away by adding a check for `max`, too, but this would be doubling down on the bug.

A better solution is to put the min and max in the same object and make this object either fully `null` or fully non-`null`:

```
function extent(nums: number[]) {
  let result: [number, number] | null = null;
  for (const num of nums) {
    if (!result) {
      result = [num, num];
    } else {
      result = [Math.min(num, result[0]), Math.max(num, result[1])];
    }
  }
  return result;
}
```

The return type is now `[number, number] | null`, which is easier for clients to work with. The min and max can be retrieved with either a non-null assertion:

```
const [min, max] = extent([0, 1, 2])!;
const span = max - min;  // OK
```

or a single check:

```
const range = extent([0, 1, 2]);
if (range) {
  const [min, max] = range;
  const span = max - min;  // OK
}
```

By using a single object to track the extent, we've improved our design, helped TypeScript understand the relationship between null values, and fixed the bug: the `if (!result)` check is now problem free.

A mix of null and non-null values can also lead to problems in classes. For instance, suppose you have a class that represents both a user and their posts on a forum:

```
class UserPosts {
  user: UserInfo | null;
  posts: Post[] | null;

  constructor() {
    this.user = null;
    this.posts = null;
  }

  async init(userId: string) {
    return Promise.all([
      async () => this.user = await fetchUser(userId),
      async () => this.posts = await fetchPostsForUser(userId)
    ]);
  }

  getUserName() {
    // ...?
  }
}
```

While the two network requests are loading, the `user` and `posts` properties will be `null`. At any time, they might both be `null`, one might be `null`, or they might both be non-`null`. There are four possibilities. This complexity will seep into every method on the class. This design is almost certain to lead to confusion, a proliferation of `null` checks, and bugs.

A better design would wait until all the data used by the class is available:

```
class UserPosts {
  user: UserInfo;
  posts: Post[];

  constructor(user: UserInfo, posts: Post[]) {
    this.user = user;
    this.posts = posts;
  }

  static async init(userId: string): Promise<UserPosts> {
    const [user, posts] = await Promise.all([
      fetchUser(userId),
      fetchPostsForUser(userId)
    ]);
    return new UserPosts(user, posts);
  }

  getUserName() {
    return this.user.name;
  }
}
```

Now the `UserPosts` class is fully non-`null`, and it's easy to write correct methods on it. Of course, if you need to perform operations while data is partially loaded, then you'll need to deal with the multiplicity of `null` and non-`null` states.

(Don't be tempted to replace nullable properties with Promises. This tends to lead to even more confusing code and forces all your methods to be async. Promises clarify the code that loads data but tend to have the opposite effect on the class that uses that data.)

Things to Remember

- Avoid designs in which one value being `null` or not `null` is implicitly related to another value being `null` or not `null`.
- Push `null` values to the perimeter of your API by making larger objects either `null` or fully non-`null`. This will make code clearer both for human readers and for the type checker.
- Consider creating a fully non-`null` class and constructing it when all values are available.
- While `strictNullChecks` may flag many issues in your code, it's indispensable for surfacing the behavior of functions with respect to null values.

Item 32: Prefer Unions of Interfaces to Interfaces of Unions

If you create an interface whose properties are union types, you should ask whether the type would make more sense as a union of more precise interfaces.

Suppose you're building a vector drawing program and want to define an interface for layers with specific geometry types:

```
interface Layer {
  layout: FillLayout | LineLayout | PointLayout;
  paint: FillPaint | LinePaint | PointPaint;
}
```

The `layout` field controls how and where the shapes are drawn (rounded corners? straight?), while the `paint` field controls styles (is the line blue? thick? thin? dashed?).

Would it make sense to have a layer whose `layout` is `LineLayout` but whose `paint` property is `FillPaint`? Probably not. Allowing this possibility makes using the library more error-prone and makes this interface difficult to work with.

A better way to model this is with separate interfaces for each type of layer:

```
interface FillLayer {
  layout: FillLayout;
  paint: FillPaint;
}
interface LineLayer {
  layout: LineLayout;
  paint: LinePaint;
}
interface PointLayer {
  layout: PointLayout;
  paint: PointPaint;
}
type Layer = FillLayer | LineLayer | PointLayer;
```

By defining `Layer` in this way, you've excluded the possibility of mixed `layout` and `paint` properties. This is an example of following Item 28's advice to prefer types that only represent valid states.

The most common example of this pattern is the "tagged union" (or "discriminated union"). In this case one of the properties is a union of string literal types:

```
interface Layer {
  type: 'fill' | 'line' | 'point';
  layout: FillLayout | LineLayout | PointLayout;
  paint: FillPaint | LinePaint | PointPaint;
}
```

As before, would it make sense to have `type: 'fill'` but then a `LineLayout` and `PointPaint`? Certainly not. Convert `Layer` to a union of interfaces to exclude this possibility:

```
interface FillLayer {
  type: 'fill';
  layout: FillLayout;
  paint: FillPaint;
}
interface LineLayer {
  type: 'line';
  layout: LineLayout;
  paint: LinePaint;
}
interface PointLayer {
  type: 'paint';
  layout: PointLayout;
  paint: PointPaint;
}
type Layer = FillLayer | LineLayer | PointLayer;
```

The `type` property is the "tag" and can be used to determine which type of `Layer` you're working with at runtime. TypeScript is also able to narrow the type of `Layer` based on the tag:

```
function drawLayer(layer: Layer) {
  if (layer.type === 'fill') {
    const {paint} = layer;  // Type is FillPaint
    const {layout} = layer;  // Type is FillLayout
  } else if (layer.type === 'line') {
    const {paint} = layer;  // Type is LinePaint
    const {layout} = layer;  // Type is LineLayout
  } else {
    const {paint} = layer;  // Type is PointPaint
    const {layout} = layer;  // Type is PointLayout
  }
}
```

By correctly modeling the relationship between the properties in this type, you help TypeScript check your code's correctness. The same code involving the initial `Layer` definition would have been cluttered with type assertions.

Because they work so well with TypeScript's type checker, tagged unions are ubiquitous in TypeScript code. Recognize this pattern and apply it when you can. If you can represent a data type in TypeScript with a tagged union, it's usually a good idea to do so. If you think of optional fields as a union of their type and `undefined`, then they fit this pattern as well. Consider this type:

```
interface Person {
  name: string;
  // These will either both be present or not be present
  placeOfBirth?: string;
  dateOfBirth?: Date;
}
```

The comment with type information is a strong sign that there might be a problem (Item 30). There is a relationship between the `placeOfBirth` and `dateOfBirth` fields that you haven't told TypeScript about.

A better way to model this is to move both of these properties into a single object. This is akin to moving `null` values to the perimeter (Item 31):

```
interface Person {
  name: string;
  birth?: {
    place: string;
    date: Date;
  }
}
```

Now TypeScript complains about values with a place but no date of birth:

```
const alanT: Person = {
  name: 'Alan Turing',
  birth: {
// ~~~~ Property 'date' is missing in type
//      '{ place: string; }' but required in type
//      '{ place: string; date: Date; }'
    place: 'London'
  }
}
```

Additionally, a function that takes a `Person` object only needs to do a single check:

```
function eulogize(p: Person) {
  console.log(p.name);
  const {birth} = p;
  if (birth) {
    console.log(`was born on ${birth.date} in ${birth.place}.`);
  }
}
```

If the structure of the type is outside your control (e.g., it's coming from an API), then you can still model the relationship between these fields using a now-familiar union of interfaces:

```
interface Name {
  name: string;
}

interface PersonWithBirth extends Name {
  placeOfBirth: string;
  dateOfBirth: Date;
}
```

```
type Person = Name | PersonWithBirth;
```

Now you get some of the same benefits as with the nested object:

```
function eulogize(p: Person) {
  if ('placeOfBirth' in p) {
    p // Type is PersonWithBirth
    const {dateOfBirth} = p  // OK, type is Date
  }
}
```

In both cases, the type definition makes the relationship between the properties more clear.

Things to Remember

- Interfaces with multiple properties that are union types are often a mistake because they obscure the relationships between these properties.
- Unions of interfaces are more precise and can be understood by TypeScript.
- Consider adding a "tag" to your structure to facilitate TypeScript's control flow analysis. Because they are so well supported, tagged unions are ubiquitous in TypeScript code.

Item 33: Prefer More Precise Alternatives to String Types

The domain of the `string` type is big: `"x"` and `"y"` are in it, but so is the complete text of *Moby Dick* (it starts `"Call me Ishmael…"` and is about 1.2 million characters long). When you declare a variable of type `string`, you should ask whether a narrower type would be more appropriate.

Suppose you're building a music collection and want to define a type for an album. Here's an attempt:

```
interface Album {
  artist: string;
  title: string;
  releaseDate: string;  // YYYY-MM-DD
  recordingType: string;  // E.g., "live" or "studio"
}
```

The prevalence of `string` types and the type information in comments (see Item 30) are strong indications that this `interface` isn't quite right. Here's what can go wrong:

```
const kindOfBlue: Album = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: 'August 17th, 1959',  // Oops!
  recordingType: 'Studio',  // Oops!
};  // OK
```

The `releaseDate` field is incorrectly formatted (according to the comment) and `"Studio"` is capitalized where it should be lowercase. But these values *are* both strings, so this object is assignable to `Album` and the type checker doesn't complain.

These broad `string` types can mask errors for valid `Album` objects, too. For example:

```
function recordRelease(title: string, date: string) { /* ... */ }
recordRelease(kindOfBlue.releaseDate, kindOfBlue.title);  // OK, should be error
```

The parameters are reversed in the call to `recordRelease` but both are strings, so the type checker doesn't complain. Because of the prevalence of `string` types, code like this is sometimes called "stringly typed."

Can you make the types narrower to prevent these sorts of issues? While the complete text of *Moby Dick* would be a ponderous artist name or album title, it's at least plausible. So `string` is appropriate for these fields. For the `releaseDate` field it's better to just use a `Date` object and avoid issues around formatting. Finally, for the `recordingType` field, you can define a union type with just two values (you could also use an `enum`, but I generally recommend avoiding these; see Item 53):

```
type RecordingType = 'studio' | 'live';

interface Album {
  artist: string;
  title: string;
  releaseDate: Date;
  recordingType: RecordingType;
}
```

With these changes TypeScript is able to do a more thorough check for errors:

```
const kindOfBlue: Album = {
  artist: 'Miles Davis',
  title: 'Kind of Blue',
  releaseDate: new Date('1959-08-17'),
  recordingType: 'Studio'
// ~~~~~~~~~~~~ Type '"Studio"' is not assignable to type 'RecordingType'
};
```

There are advantages to this approach beyond stricter checking. First, explicitly defining the type ensures that its meaning won't get lost as it's passed around. If you wanted to find albums of just a certain recording type, for instance, you might define a function like this:

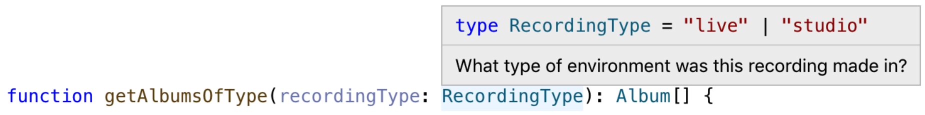
```
function getAlbumsOfType(recordingType: string): Album[] {
  // ...
}
```

How does the caller of this function know what `recordingType` is expected to be? It's just a `string`. The comment explaining that it's `"studio"` or `"live"` is hidden in the definition of `Album`, where the user might not think to look.

Second, explicitly defining a type allows you attach documentation to it (see Item 48):

```
/** What type of environment was this recording made in?  */
type RecordingType = 'live' | 'studio';
```

When you change `getAlbumsOfType` to take a `RecordingType`, the caller is able to click through and see the documentation (see Figure 4-1).



```
                                   type RecordingType = "live" | "studio"
                                   What type of environment was this recording made in?
function getAlbumsOfType(recordingType: RecordingType): Album[] {
```

Figure 4-1. Using a named type instead of string makes it possible to attach documentation to the type that is surfaced in your editor.

Another common misuse of `string` is in function parameters. Say you want to write a function that pulls out all the values for a single field in an array. The Underscore library calls this "pluck":

```
function pluck(records, key) {
  return record.map(record => record[key]);
}
```

How would you type this? Here's an initial attempt:

```
function pluck(record: any[], key: string): any[] {
  return record.map(r => r[key]);
}
```

This type checks but isn't great. The `any` types are problematic, particularly on the return value (see Item 38). The first step to improving the type signature is introducing a generic type parameter:

```
function pluck<T>(record: T[], key: string): any[] {
  return record.map(r => r[key]);
                      // ~~~~~~ Element implicitly has an 'any' type
                      //        because type '{}' has no index signature
}
```

TypeScript is now complaining that the `string` type for `key` is too broad. And it's right to do so: if you pass in an array of `Album`s then there are only four valid values for `key` ("artist," "title," "releaseDate," and "recordingType"), as opposed to the vast set of strings. This is precisely what the `keyof Album` type is:

```
type K = keyof Album;
// Type is "artist" | "title" | "releaseDate" | "recordingType"
```

So the fix is to replace `string` with `keyof T`:

```
function pluck<T>(record: T[], key: keyof T) {
  return record.map(r => r[key]);
}
```

This passes the type checker. We've also let TypeScript infer the return type. How does it do? If you mouse over `pluck` in your editor, the inferred type is:

```
function pluck<T>(record: T[], key: keyof T): T[keyof T][]
```

`T[keyof T]` is the type of any possible value in `T`. If you're passing in a single string as the `key`, this is too broad. For example:

```
const releaseDates = pluck(albums, 'releaseDate'); // Type is (string | Date)[]
```

The type should be `Date[]`, not `(string | Date)[]`. While `keyof T` is much narrower than `string`, it's *still* too broad. To narrow it further, we need to introduce a second generic parameter that is a subset of `keyof T` (probably a single value):

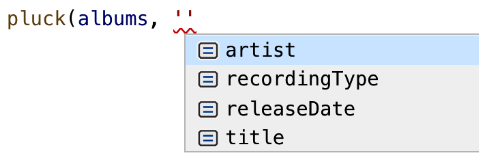
```
function pluck<T, K extends keyof T>(record: T[], key: K): T[K][] {
  return record.map(r => r[key]);
}
```

(For more on `extends` in this context, see Item 14.)

The type signature is now completely correct. We can check this by calling `pluck` in a few different ways:

```
pluck(albums, 'releaseDate'); // Type is Date[]
pluck(albums, 'artist');  // Type is string[]
pluck(albums, 'recordingType');  // Type is RecordingType[]
pluck(albums, 'recordingDate');
           // ~~~~~~~~~~~~~~~ Argument of type '"recordingDate"' is not
           //                 assignable to parameter of type ...
```

The language service is even able to offer autocomplete on the keys of `Album` (as shown in Figure 4-2).



```
pluck(albums, '')
```

Figure 4-2. Using a parameter type of keyof Album instead of string results in better autocomplete in your editor.

`string` has some of the same problems as `any`: when used inappropriately, it permits invalid values and hides relationships between types. This thwarts the type checker and can hide real bugs. TypeScript's ability to define subsets of `string` is a powerful way to bring type safety to JavaScript code. Using more precise types will both catch errors and improve the readability of your code.

Things to Remember

- Avoid "stringly typed" code. Prefer more appropriate types where not every `string` is a possibility.
- Prefer a union of string literal types to `string` if that more accurately describes the domain of a variable. You'll get stricter type checking and improve the development experience.
- Prefer `keyof T` to `string` for function parameters that are expected to be properties of an object.

Item 34: Prefer Incomplete Types to Inaccurate Types

In writing type declarations you'll inevitably find situations where you can model behavior in a more precise or less precise way. Precision in types is generally a good thing because it will help your users catch bugs and take advantage of the tooling that TypeScript provides. But take care as you increase the precision of your type declarations: it's easy to make mistakes, and incorrect types can be worse than no types at all.

Suppose you are writing type declarations for GeoJSON, a format we've seen before in Item 31. A GeoJSON Geometry can be one of a few types, each of which have differently shaped coordinate arrays:

```
interface Point {
  type: 'Point';
  coordinates: number[];
}
interface LineString {
  type: 'LineString';
  coordinates: number[][];
}
interface Polygon {
  type: 'Polygon';
  coordinates: number[][][];
}
type Geometry = Point | LineString | Polygon;  // Also several others
```

This is fine, but `number[]` for a coordinate is a bit imprecise. Really these are latitudes and longitudes, so perhaps a tuple type would be better:

```
type GeoPosition = [number, number];
interface Point {
  type: 'Point';
  coordinates: GeoPosition;
}
// Etc.
```

You publish your more precise types to the world and wait for the adulation to roll in. Unfortunately, a user complains that your new types have broken everything. Even though you've only ever used latitude and longitude, a position in GeoJSON is allowed to have a third element, an elevation, and potentially more. In an attempt to make the type declarations more precise, you've gone too far and made the types inaccurate! To continue using your type declarations, your user will have to introduce type assertions or silence the type checker entirely with `as any`.

As another example, consider trying to write type declarations for a Lisp-like language defined in JSON:

```
12
"red"
["+", 1, 2]  // 3
["/", 20, 2]  // 10
["case", [">", 20, 10], "red", "blue"]  // "red"
["rgb", 255, 0, 127]  // "#FF007F"
```

The Mapbox library uses a system like this to determine the appearance of map features across many devices. There's a whole spectrum of precision with which you could try to type this:

1. Allow anything.
2. Allow strings, numbers, and arrays.
3. Allow strings, numbers, and arrays starting with known function names.
4. Make sure each function gets the correct number of arguments.
5. Make sure each function gets the correct type of arguments.

The first two options are straightforward:

```
type Expression1 = any;
type Expression2 = number | string | any[];
```

Beyond this, you should introduce a test set of expressions that are valid and expressions that are not. As you make your types more precise, this will help prevent regressions (see Item 52):

```
const tests: Expression2[] = [
  10,
  "red",
  true,
// ~~~ Type 'true' is not assignable to type 'Expression2'
  ["+", 10, 5],
  ["case", [">", 20, 10], "red", "blue", "green"],  // Too many values
  ["**", 2, 31],  // Should be an error: no "**" function
  ["rgb", 255, 128, 64],
```

```
    ["rgb", 255, 0, 127, 0]  // Too many values
  ];
```

To go to the next level of precision you can use a union of string literal types as the first element of a tuple:

```
type FnName = '+' | '-' | '*' | '/' | '>' | '<' | 'case' | 'rgb';
type CallExpression = [FnName, ...any[]];
type Expression3 = number | string | CallExpression;

const tests: Expression3[] = [
  10,
  "red",
  true,
// ~~~ Type 'true' is not assignable to type 'Expression3'
  ["+", 10, 5],
  ["case", [">", 20, 10], "red", "blue", "green"],
  ["**", 2, 31],
// ~~~~~~~~~~~ Type '"**"' is not assignable to type 'FnName'
  ["rgb", 255, 128, 64]
];
```

There's one new caught error and no regressions. Pretty good!

What if you want to make sure that each function gets the correct number of arguments? This gets trickier since the type now needs to be recursive to reach down into all the function calls. As of TypeScript 3.6, to make this work you needed to introduce at least one `interface`. Since `interface`s can't be unions, you'll have to write the call expressions using `interface` instead. This is a bit awkward since fixed-length arrays are most easily expressed as tuple types. But you *can* do it:

```
type Expression4 = number | string | CallExpression;

type CallExpression = MathCall | CaseCall | RGBCall;

interface MathCall {
  0: '+' | '-' | '/' | '*' | '>' | '<';
  1: Expression4;
  2: Expression4;
  length: 3;
}

interface CaseCall {
  0: 'case';
  1: Expression4;
  2: Expression4;
  3: Expression4;
  length: 4 | 6 | 8 | 10 | 12 | 14 | 16 // etc.
}

interface RGBCall {
  0: 'rgb';
```

```
  1: Expression4;
  2: Expression4;
  3: Expression4;
  length: 4;
}

const tests: Expression4[] = [
  10,
  "red",
  true,
// ~~~ Type 'true' is not assignable to type 'Expression4'
  ["+", 10, 5],
  ["case", [">", 20, 10], "red", "blue", "green"],
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  Type '["case", [">", ...], ...]' is not assignable to type 'string'
  ["**", 2, 31],
// ~~~~~~~~~~~~ Type '["**", number, number]' is not assignable to type 'string
  ["rgb", 255, 128, 64],
  ["rgb", 255, 128, 64, 73]
// ~~~~~~~~~~~~~~~~~~~~~~~~ Type '["rgb", number, number, number, number]'
//                          is not assignable to type 'string'
];
```

Now all the invalid expressions produce errors. And it's interesting that you can express something like "an array of even length" using a TypeScript `interface`. But these error messages aren't very good, and the error about `**` has gotten quite a bit worse since the previous typings.

Is this an improvement over the previous, less precise types? The fact that you get errors for some incorrect usages is a win, but the errors will make this type more difficult to work with. Language services are as much a part of the TypeScript experience as type checking (see Item 6), so it's a good idea to look at the error messages resulting from your type declarations and try autocomplete in situations where it should work. If your new type declarations are more precise but break autocomplete, then they'll make for a less enjoyable TypeScript development experience.

The complexity of this type declaration has also increased the odds that a bug will creep in. For example, `Expression4` requires that all math operators take two parameters, but the Mapbox expression spec says that `+` and `*` can take more. Also, `-` can take a single parameter, in which case it negates its input. `Expression4` incorrectly flags errors in all of these:

```
 const okExpressions: Expression4[] = [
   ['-', 12],
// ~~~~~~~~~ Type '["-", number]' is not assignable to type 'string'
   ['+', 1, 2, 3],
// ~~~~~~~~~~~~~~ Type '["+", number, ...]' is not assignable to type 'string'
   ['*', 2, 3, 4],
// ~~~~~~~~~~~~~~ Type '["*", number, ...]' is not assignable to type 'string'
 ];
```

Once again, in trying to be more precise we've overshot and become inaccurate. These inaccuracies can be corrected, but you'll want to expand your test set to convince yourself that you haven't missed anything else. Complex code generally requires more tests, and the same is true of types.

As you refine types, it can be helpful to think of the "uncanny valley" metaphor. Refining very imprecise types like `any` is usually helpful. But as your types get more precise, the expectation that they'll also be accurate increases. You'll start to rely on the types more, and so inaccuracies will produce bigger problems.

Things to Remember

- Avoid the uncanny valley of type safety: incorrect types are often worse than no types.
- If you cannot model a type accurately, do not model it inaccurately! Acknowledge the gaps using `any` or `unknown`.
- Pay attention to error messages and autocomplete as you make typings increasingly precise. It's not just about correctness: developer experience matters, too.

Item 35: Generate Types from APIs and Specs, Not Data

The other items in this chapter have discussed the many benefits of designing your types well and shown what can go wrong if you don't. A well-designed type makes TypeScript a pleasure to use, while a poorly designed one can make it miserable. But this does put quite a bit of pressure on type design. Wouldn't it be nice if you didn't have to do this yourself?

At least some of your types are likely to come from outside your program: file formats, APIs, or specs. In these cases you may be able to avoid writing types by generating them instead. If you do this, the key is to generate types from specifications, rather than from example data. When you generate types from a spec, TypeScript will help ensure that you haven't missed any cases. When you generate types from data, you're only considering the examples you've seen. You might be missing important edge cases that could break your program.

In Item 31 we wrote a function to calculate the bounding box of a GeoJSON Feature. Here's what it looked like:

```
function calculateBoundingBox(f: GeoJSONFeature): BoundingBox | null {
  let box: BoundingBox | null = null;

  const helper = (coords: any[]) => {
    // ...
  };
  const {geometry} = f;
  if (geometry) {
    helper(geometry.coordinates);
  }

  return box;
}
```

The `GeoJSONFeature` type was never explicitly defined. You could write it using some of the examples from Item 31. But a better approach is to use the formal GeoJSON spec.[1] Fortunately for us, there are already TypeScript type declarations for it on DefinitelyTyped. You can add these in the usual way:

```
$ npm install --save-dev @types/geojson
+ @types/geojson@7946.0.7
```

When you plug in the GeoJSON declarations, TypeScript immediately flags an error:

```
import {Feature} from 'geojson';

function calculateBoundingBox(f: Feature): BoundingBox | null {
  let box: BoundingBox | null = null;

  const helper = (coords: any[]) => {
    // ...
  };

  const {geometry} = f;
  if (geometry) {
    helper(geometry.coordinates);
                 // ~~~~~~~~~~~
                 // Property 'coordinates' does not exist on type 'Geometry'
                 //   Property 'coordinates' does not exist on type
                 //   'GeometryCollection'
  }

  return box;
}
```

The problem is that your code assumes a geometry will have a `coordinates` property. This is true for many geometries, including points, lines, and polygons. But a GeoJSON geometry can also be a `GeometryCollection`, a heterogeneous collection of other geometries. Unlike the other geometry types, it does not have a `coordinates` property.

1 GeoJSON is also known as RFC 7946. The very readable spec is at *http://geojson.org*.

If you call `calculateBoundingBox` on a Feature whose geometry is a `GeometryCollection`, it will throw an error about not being able to read property `0` of `undefined`. This is a real bug! And we caught it using type definitions from a spec.

One option for fixing it is to explicitly disallow `GeometryCollections`, as shown here:

```
const {geometry} = f;
if (geometry) {
  if (geometry.type === 'GeometryCollection') {
    throw new Error('GeometryCollections are not supported.');
  }
  helper(geometry.coordinates);  // OK
}
```

TypeScript is able to refine the type of `geometry` based on the check, so the reference to `geometry.coordinates` is allowed. If nothing else, this results in a clearer error message for the user.

But the better solution is to support all the types of geometry! You can do this by pulling out another helper function:

```
const geometryHelper = (g: Geometry) => {
  if (geometry.type === 'GeometryCollection') {
    geometry.geometries.forEach(geometryHelper);
  } else {
    helper(geometry.coordinates);  // OK
  }
}

const {geometry} = f;
if (geometry) {
  geometryHelper(geometry);
}
```

Had you written type declarations for GeoJSON yourself, you would have based them off of your understanding and experience with the format. This might not have included `GeometryCollections` and would have led to a false sense of security about your code's correctness. Using types based on a spec gives you confidence that your code will work with all values, not just the ones you've seen.

Similar considerations apply to API calls: if you can generate types from the specification of an API, then it is usually a good idea to do so. This works particularly well with APIs that are typed themselves, such as GraphQL.

A GraphQL API comes with a schema that specifies all the possible queries and interfaces using a type system somewhat similar to TypeScript. You write queries that request specific fields in these interfaces. For example, to get information about a repository using the GitHub GraphQL API you might write:

```
query {
  repository(owner: "Microsoft", name: "TypeScript") {
    createdAt
    description
  }
}
```

The result is:

```
{
  "data": {
    "repository": {
      "createdAt": "2014-06-17T15:28:39Z",
      "description":
        "TypeScript is a superset of JavaScript that compiles to JavaScript."
    }
  }
}
```

The beauty of this approach is that you can generate TypeScript types *for your specific query*. As with the GeoJSON example, this helps ensure that you model the relationships between types and their nullability accurately.

Here's a query to get the open source license for a GitHub repository:

```
query getLicense($owner:String!, $name:String!){
  repository(owner:$owner, name:$name) {
    description
    licenseInfo {
      spdxId
      name
    }
  }
}
```

`$owner` and `$name` are GraphQL variables which are themselves typed. The type syntax is similar enough to TypeScript that it can be confusing to go back and forth. `String` is a GraphQL type—it would be `string` in TypeScript (see Item 10). And while TypeScript types are not nullable, types in GraphQL are. The `!` after the type indicates that it is guaranteed to not be null.

There are many tools to help you go from a GraphQL query to TypeScript types. One is Apollo. Here's how you use it:

```
$ apollo client:codegen \
    --endpoint https://api.github.com/graphql \
    --includes license.graphql \
    --target typescript
Loading Apollo Project
Generating query files with 'typescript' target - wrote 2 files
```

You need a GraphQL schema to generate types for a query. Apollo gets this from the `api.github.com/graphql` endpoint. The output looks like this:

```
export interface getLicense_repository_licenseInfo {
  __typename: "License";
  /** Short identifier specified by <https://spdx.org/licenses> */
  spdxId: string | null;
  /** The license full name specified by <https://spdx.org/licenses> */
  name: string;
}

export interface getLicense_repository {
  __typename: "Repository";
  /** The description of the repository. */
  description: string | null;
  /** The license associated with the repository */
  licenseInfo: getLicense_repository_licenseInfo | null;
}

export interface getLicense {
  /** Lookup a given repository by the owner and repository name. */
  repository: getLicense_repository | null;
}

export interface getLicenseVariables {
  owner: string;
  name: string;
}
```

The important bits to note here are that:

- Interfaces are generated for both the query parameters (`getLicenseVariables`) and the response (`getLicense`).
- Nullability information is transferred from the schema to the response interfaces. The `repository`, `description`, `licenseInfo`, and `spdxId` fields are nullable, whereas the license `name` and the query variables are not.
- Documentation is transferred as JSDoc so that it appears in your editor (Item 48). These comments come from the GraphQL schema itself.

This type information helps ensure that you use the API correctly. If your queries change, the types will change. If the schema changes, then so will your types. There is no risk that your types and reality diverge since they are both coming from a single source of truth: the GraphQL schema.

What if there's no spec or official schema available? Then you'll have to generate types from data. Tools like `quicktype` can help with this. But be aware that your types may not match reality: there may be edge cases that you've missed.

Even if you're not aware of it, you are already benefiting from code generation. TypeScript's type declarations for the browser DOM API are generated from the official interfaces (see Item 55). This ensures that they correctly model a complicated system and helps TypeScript catch errors and misunderstandings in your own code.

Things to Remember

- Consider generating types for API calls and data formats to get type safety all the way to the edge of your code.
- Prefer generating code from specs rather than data. Rare cases matter!

Item 36: Name Types Using the Language of Your Problem Domain

> There are only two hard problems in Computer Science: cache invalidation and naming things.
>
> —Phil Karlton

This book has had much to say about the *shape* of types and the sets of values in their domains, but much less about what you *name* your types. But this is an important part of type design, too. Well-chosen type, property, and variable names can clarify intent and raise the level of abstraction of your code and types. Poorly chosen types can obscure your code and lead to incorrect mental models.

Suppose you're building out a database of animals. You create an interface to represent one:

```
interface Animal {
  name: string;
  endangered: boolean;
  habitat: string;
}

const leopard: Animal = {
  name: 'Snow Leopard',
  endangered: false,
  habitat: 'tundra',
};
```

There are a few issues here:

- `name` is a very general term. What sort of name are you expecting? A scientific name? A common name?
- The boolean `endangered` field is also ambiguous. What if an animal is extinct? Is the intent here "endangered or worse?" Or does it literally mean endangered?

- The `habitat` field is very ambiguous, not just because of the overly broad `string` type (Item 33) but also because it's unclear what's meant by "habitat."
- The variable name is `leopard`, but the value of the `name` property is "Snow Leopard." Is this distinction meaningful?

Here's a type declaration and value with less ambiguity:

```
interface Animal {
  commonName: string;
  genus: string;
  species: string;
  status: ConservationStatus;
  climates: KoppenClimate[];
}
type ConservationStatus = 'EX' | 'EW' | 'CR' | 'EN' | 'VU' | 'NT' | 'LC';
type KoppenClimate = |
  'Af' | 'Am' | 'As' | 'Aw' |
  'BSh' | 'BSk' | 'BWh' | 'BWk' |
  'Cfa' | 'Cfb' | 'Cfc' | 'Csa' | 'Csb' | 'Csc' | 'Cwa' | 'Cwb' | 'Cwc' |
  'Dfa' | 'Dfb' | 'Dfc' | 'Dfd' |
  'Dsa' | 'Dsb' | 'Dsc' | 'Dwa' | 'Dwb' | 'Dwc' | 'Dwd' |
  'EF' | 'ET';
const snowLeopard: Animal = {
  commonName: 'Snow Leopard',
  genus: 'Panthera',
  species: 'Uncia',
  status: 'VU',  // vulnerable
  climates: ['ET', 'EF', 'Dfd'],  // alpine or subalpine
};
```

This makes a number of improvements:

- `name` has been replaced with more specific terms: `commonName`, `genus`, and `species`.
- `endangered` has become `conservationStatus` and uses a standard classification system from the IUCN.
- `habitat` has become `climates` and uses another standard taxonomy, the Köppen climate classification.

If you needed more information about the fields in the first version of this type, you'd have to go find the person who wrote them and ask. In all likelihood, they've left the company or don't remember. Worse yet, you might run `git blame` to find out who wrote these lousy types, only to find that it was you!

The situation is much improved with the second version. If you want to learn more about the Köppen climate classification system or track down what the precise meaning of a conservation status is, then there are myriad resources online to help you.

Every domain has specialized vocabulary to describe its subject. Rather than inventing your own terms, try to reuse terms from the domain of your problem. These vocabularies have often been honed over years, decades, or centuries and are well understood by people in the field. Using these terms will help you communicate with users and increase the clarity of your types.

Take care to use domain vocabulary accurately: co-opting the language of a domain to mean something different is even more confusing than inventing your own.

Here are a few other rules to keep in mind as you name types, properties, and variables:

- Make distinctions meaningful. In writing and speech it can be tedious to use the same word over and over. We introduce synonyms to break the monotony. This makes prose more enjoyable to read, but it has the opposite effect on code. If you use two different terms, make sure you're drawing a meaningful distinction. If not, you should use the same term.
- Avoid vague, meaningless names like "data," "info," "thing," "item," "object," or the ever-popular "entity." If Entity has a specific meaning in your domain, fine. But if you're using it because you don't want to think of a more meaningful name, then you'll eventually run into trouble.
- Name things for what they are, not for what they contain or how they are computed. `Directory` is more meaningful than `INodeList`. It allows you to think about a directory as a concept, rather than in terms of its implementation. Good names can increase your level of abstraction and decrease your risk of inadvertent collisions.

Things to Remember

- Reuse names from the domain of your problem where possible to increase the readability and level of abstraction of your code.
- Avoid using different names for the same thing: make distinctions in names meaningful.

Item 37: Consider "Brands" for Nominal Typing

Item 4 discussed structural ("duck") typing and how it can sometimes lead to surprising results:

```
interface Vector2D {
  x: number;
  y: number;
}
function calculateNorm(p: Vector2D) {
  return Math.sqrt(p.x * p.x + p.y * p.y);
}

calculateNorm({x: 3, y: 4});  // OK, result is 5
const vec3D = {x: 3, y: 4, z: 1};
calculateNorm(vec3D);  // OK! result is also 5
```

What if you'd like `calculateNorm` to reject 3D vectors? This goes against the structural typing model of TypeScript but is certainly more mathematically correct.

One way to achieve this is with *nominal typing*. With nominal typing, a value is a `Vector2D` because you say it is, not because it has the right shape. To approximate this in TypeScript, you can introduce a "brand" (think cows, not Coca-Cola):

```
interface Vector2D {
  _brand: '2d';
  x: number;
  y: number;
}
function vec2D(x: number, y: number): Vector2D {
  return {x, y, _brand: '2d'};
}
function calculateNorm(p: Vector2D) {
  return Math.sqrt(p.x * p.x + p.y * p.y);  // Same as before
}

calculateNorm(vec2D(3, 4)); // OK, returns 5
const vec3D = {x: 3, y: 4, z: 1};
calculateNorm(vec3D);
           // ~~~~~ Property '_brand' is missing in type...
```

The brand ensures that the vector came from the right place. Granted there's nothing stopping you from adding `_brand: '2d'` to the `vec3D` value. But this is moving from the accidental into the malicious. This sort of brand is typically enough to catch inadvertent misuses of functions.

Interestingly, you can get many of the same benefits as explicit brands while operating only in the type system. This removes runtime overhead and also lets you brand built-in types like `string` or `number` where you can't attach additional properties.

For instance, what if you have a function that operates on the filesystem and requires an absolute (as opposed to a relative) path? This is easy to check at runtime (does the path start with "/"?) but not so easy in the type system.

Here's an approach with brands:

```
type AbsolutePath = string & {_brand: 'abs'};
function listAbsolutePath(path: AbsolutePath) {
  // ...
}
```

```
function isAbsolutePath(path: string): path is AbsolutePath {
  return path.startsWith('/');
}
```

You can't construct an object that is a `string` and has a `_brand` property. This is purely a game with the type system.

If you have a `string` path that could be either absolute or relative, you can check using the type guard, which will refine its type:

```
function f(path: string) {
  if (isAbsolutePath(path)) {
    listAbsolutePath(path);
  }
  listAbsolutePath(path);
                // ~~~~ Argument of type 'string' is not assignable
                //      to parameter of type 'AbsolutePath'
}
```

This sort of approach could be helpful in documenting which functions expect absolute or relative paths and which type of path each variable holds. It is not an ironclad guarantee, though: `path as AbsolutePath` will succeed for any `string`. But if you avoid these sorts of assertions, then the only way to get an `AbsolutePath` is to be given one or to check, which is exactly what you want.

This approach can be used to model many properties that cannot be expressed within the type system. For example, using binary search to find an element in a list:

```
function binarySearch<T>(xs: T[], x: T): boolean {
  let low = 0, high = xs.length - 1;
  while (high >= low) {
    const mid = low + Math.floor((high - low) / 2);
    const v = xs[mid];
    if (v === x) return true;
    [low, high] = x > v ? [mid + 1, high] : [low, mid - 1];
  }
  return false;
}
```

This works if the list is sorted, but will result in false negatives if it is not. You can't represent a sorted list in TypeScript's type system. But you can create a brand:

```
type SortedList<T> = T[] & {_brand: 'sorted'};

function isSorted<T>(xs: T[]): xs is SortedList<T> {
  for (let i = 1; i < xs.length; i++) {
    if (xs[i] > xs[i - 1]) {
      return false;
    }
  }
  return true;
}
```

```
function binarySearch<T>(xs: SortedList<T>, x: T): boolean {
  // ...
}
```

In order to call this version of `binarySearch`, you either need to be given a `SortedList` (i.e., have a proof that the list is sorted) or prove that it's sorted yourself using `isSorted`. The linear scan isn't great, but at least you'll be safe!

This is a helpful perspective to have on the type checker in general. In order to call a method on an object, for instance, you either need to be given a non-`null` object or prove that it's non-`null` yourself with a conditional.

You can also brand `number` types—for example, to attach units:

```
type Meters = number & {_brand: 'meters'};
type Seconds = number & {_brand: 'seconds'};

const meters = (m: number) => m as Meters;
const seconds = (s: number) => s as Seconds;

const oneKm = meters(1000);  // Type is Meters
const oneMin = seconds(60);  // Type is Seconds
```

This can be awkward in practice since arithmetic operations make the numbers forget their brands:

```
const tenKm = oneKm * 10;  // Type is number
const v = oneKm / oneMin;  // Type is number
```

If your code involves lots of numbers with mixed units, however, this may still be an attractive approach to documenting the expected types of numeric parameters.

Things to Remember

- TypeScript uses structural ("duck") typing, which can sometimes lead to surprising results. If you need nominal typing, consider attaching "brands" to your values to distinguish them.
- In some cases you may be able to attach brands entirely in the type system, rather than at runtime. You can use this technique to model properties outside of TypeScript's type system.