```
function foo(abc: ABC) {
  for (const [k, v] of Object.entries(abc)) {
    k  // Type is string
    v  // Type is any
  }
}
```

While these types may be hard to work with, they are at least honest!

You should also be aware of the possibility of *prototype pollution*. Even in the case of an object literal that you define, for-in can produce additional keys:

```
> Object.prototype.z = 3; // Please don't do this!
> const obj = {x: 1, y: 2};
> for (const k in obj) { console.log(k); }
x
y
z
```

Hopefully this doesn't happen in a nonadversarial environment (you should never add enumerable properties to `Object.prototype`), but it is another reason that for-in produces `string` keys even for object literals.

If you want to iterate over the keys and values in an object, use either a `keyof` declaration (`let k: keyof T`) or `Object.entries`. The former is appropriate for constants or other situations where you know that the object won't have additional keys and you want precise types. The latter is more generally appropriate, though the key and value types are more difficult to work with.

Things to Remember

- Use `let k: keyof T` and a for-in loop to iterate objects when you know exactly what the keys will be. Be aware that any objects your function receives as parameters might have additional keys.
- Use `Object.entries` to iterate over the keys and values of any object.

Item 55: Understand the DOM hierarchy

Most of the items in this book are agnostic about where you run your TypeScript: in a web browser, on a server, on a phone. This one is different. If you're not working in a browser, skip ahead!

The DOM hierarchy is always present when you're running JavaScript in a web browser. When you use `document.getElementById` to get an element or `document.createElement` to create one, it's always a particular kind of element, even if you're not entirely familiar with the taxonomy. You call the methods and use the properties that you want and hope for the best.

With TypeScript, the hierarchy of DOM elements becomes more visible. Knowing your `Node`s from your `Element`s and `EventTarget`s will help you debug type errors and decide when type assertions are appropriate. Because so many APIs are based on the DOM, this is relevant even if you're using a framework like React or d3.

Suppose you want to track a user's mouse as they drag it across a `<div>`. You write some seemingly innocuous JavaScript:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
  const dragStart = [eDown.clientX, eDown.clientY];
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
    targetEl.removeEventListener('mouseup', handleUp);
    const dragEnd = [eUp.clientX, eUp.clientY];
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
}
const div = document.getElementById('surface');
div.addEventListener('mousedown', handleDrag);
```

TypeScript's type checker flags no fewer than 11 errors in these 14 lines of code:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~~~           Object is possibly 'null'.
//         ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
  const dragStart = [
     eDown.clientX, eDown.clientY];
        // ~~~~~~~                Property 'clientX' does not exist on 'Event'
        //                ~~~~~~~ Property 'clientY' does not exist on 'Event'
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
//  ~~~~~~~~           Object is possibly 'null'.
//           ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
    targetEl.removeEventListener('mouseup', handleUp);
//  ~~~~~~~~ Object is possibly 'null'
    const dragEnd = [
       eUp.clientX, eUp.clientY];
        // ~~~~~~~                Property 'clientX' does not exist on 'Event'
        //              ~~~~~~~   Property 'clientY' does not exist on 'Event'
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
// ~~~~~~~ Object is possibly 'null'
}
```

```
    const div = document.getElementById('surface');
    div.addEventListener('mousedown', handleDrag);
 // ~~~ Object is possibly 'null'
```

What went wrong? What's this `EventTarget`? And why might everything be `null`?

To understand the `EventTarget` errors it helps to dig into the DOM hierarchy a bit. Here's some HTML:

```
<p id="quote">and <i>yet</i> it moves</p>
```

If you open your browser's JavaScript console and get a reference to the `p` element, you'll see that it's an `HTMLParagraphElement`:

```
const p = document.getElementsByTagName('p')[0];
p instanceof HTMLParagraphElement
// True
```

An `HTMLParagraphElement` is a subtype of `HTMLElement`, which is a subtype of `Element`, which is a subtype of `Node`, which is a subtype of `EventTarget`. Here are some examples of types along the hierarchy:

Table 7-1. Types in the DOM Hierarchy

Type	Examples
EventTarget	`window`, `XMLHttpRequest`
Node	`document`, `Text`, `Comment`
Element	*includes HTMLElements, SVGElements*
HTMLElement	`<i>`, `<b>`
HTMLButtonElement	`<button>`

An `EventTarget` is the most generic of DOM types. All you can do with it is add event listeners, remove them, and dispatch events. With this in mind, the `classList` errors start to make a bit more sense:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~~~           Object is possibly 'null'
//         ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
  // ...
}
```

As its name implies, an `Event`'s `currentTarget` property is an `EventTarget`. It could even be `null`. TypeScript has no reason to believe that it has a `classList` property. While an `EventTargets` *could* be an `HTMLElement` in practice, from the type system's perspective there's no reason it couldn't be `window` or `XMLHTTPRequest`.

Moving up the hierarchy we come to `Node`. A couple of examples of `Node`s that are not `Element`s are text fragments and comments. For instance, in this HTML:

```
<p>
  And <i>yet</i> it moves
  <!-- quote from Galileo -->
</p>
```

the outermost element is an `HTMLParagraphElement`. As you can see here, it has `children` and `childNodes`:

```
> p.children
HTMLCollection [i]
> p.childNodes
NodeList(5) [text, i, text, comment, text]
```

`children` returns an `HTMLCollection`, an array-like structure containing just the child `Element`s (`<i>yet</i>`). `childNodes` returns a `NodeList`, an Array-like collection of `Node`s. This includes not just `Element`s (`<i>yet</i>`) but also text fragments ("And," "it moves") and comments ("quote from Galileo").

What's the difference between an `Element` and an `HTMLElement`? There are non-HTML `Element`s including the whole hierarchy of SVG tags. These are `SVGElement`s, which are another type of `Element`. What's the type of an `<html>` or `<svg>` tag? They're `HTMLHtmlElement` and `SVGSvgElement`.

Sometimes these specialized classes will have properties of their own—for example, an `HTMLImageElement` has a `src` property, and an `HTMLInputElement` has a `value` property. If you want to read one of these properties off a value, its type must be specific enough to have that property.

TypeScript's type declarations for the DOM make liberal use of literal types to try to get you the most specific type possible. For example:

```
document.getElementsByTagName('p')[0];  // HTMLParagraphElement
document.createElement('button');  // HTMLButtonElement
document.querySelector('div');  // HTMLDivElement
```

but this is not always possible, notably with `document.getElementById`:

```
document.getElementById('my-div');  // HTMLElement
```

While type assertions are generally frowned upon (Item 9), this is a case where you know more than TypeScript does and so they are appropriate. There's nothing wrong with this, so long as you know that `#my-div` is a div:

```
document.getElementById('my-div') as HTMLDivElement;
```

with `strictNullChecks` enabled, you will need to consider the case that `document.getElementById` returns `null`. Depending on whether this can really happen, you can either add an if statement or an assertion (`!`):

```
const div = document.getElementById('my-div')!;
```

These types are not specific to TypeScript. Rather, they are generated from the formal specification of the DOM. This is an example of the advice of Item 35 to generate types from specs when possible.

So much for the DOM hierarchy. What about the `clientX` and `clientY` errors?

```
function handleDrag(eDown: Event) {
  // ...
  const dragStart = [
     eDown.clientX, eDown.clientY];
        // ~~~~~~~                Property 'clientX' does not exist on 'Event'
        //                ~~~~~~~ Property 'clientY' does not exist on 'Event'
  // ...
}
```

In addition to the hierarchy for `Node`s and `Element`s, there is also a hierarchy for `Event`s. The Mozilla documentation currently lists no fewer than 52 types of `Event`!

Plain `Event` is the most generic type of event. More specific types include:

`UIEvent`
: Any sort of user interface event

`MouseEvent`
: An event triggered by the mouse such as a click

`TouchEvent`
: A touch event on a mobile device

`WheelEvent`
: An event triggered by rotating the scroll wheel

`KeyboardEvent`
: A key press

The problem in `handleDrag` is that the events are declared as `Event`, while `clientX` and `clientY` exist only on the more specific `MouseEvent` type.

So how can you fix the example from the start of this item? TypeScript's declarations for the DOM make extensive use of context (Item 26). Inlining the mousedown handler gives TypeScript more information to work with and removes most of the errors. You can also declare the parameter type to be `MouseEvent` rather than `Event`. Here's a version that uses both techniques to fix the errors:

```
function addDragHandler(el: HTMLElement) {
  el.addEventListener('mousedown', eDown => {
    const dragStart = [eDown.clientX, eDown.clientY];
    const handleUp = (eUp: MouseEvent) => {
      el.classList.remove('dragging');
```

```
      el.removeEventListener('mouseup', handleUp);
      const dragEnd = [eUp.clientX, eUp.clientY];
      console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
    }
    el.addEventListener('mouseup', handleUp);
  });
}

const div = document.getElementById('surface');
if (div) {
  addDragHandler(div);
}
```

The `if` statement at the end handles the possibility that there is no `#surface` element. If you know that this element exists, you could use an assertion instead (`div!`). `addDragHandler` requires a non-null `HTMLElement`, so this is an example of pushing `null` values to the perimeter (Item 31).

Things to Remember

- The DOM has a type hierarchy that you can usually ignore while writing JavaScript. But these types become more important in TypeScript. Understanding them will help you write TypeScript for the browser.
- Know the differences between `Node`, `Element`, `HTMLElement`, and `EventTarget`, as well as those between `Event` and `MouseEvent`.
- Either use a specific enough type for DOM elements and Events in your code or give TypeScript the context to infer it.

Item 56: Don't Rely on Private to Hide Information

JavaScript has historically lacked a way to make properties of a class private. The usual workaround is a convention of prefixing fields that are not part of a public API with underscores:

```
class Foo {
  _private = 'secret123';
}
```

But this only discourages users from accessing private data. It is easy to circumvent:

```
const f = new Foo();
f._private;  // 'secret123'
```

TypeScript adds `public`, `protected`, and `private` field modifiers that seem to provide some enforcement: