CHAPTER 1
Getting to Know TypeScript

Before we dive into the details, this chapter helps you understand the big picture of TypeScript. What is it and how should you think about it? How does it relate to JavaScript? Are its types nullable or are they not? What's this about `any`? And ducks?

TypeScript is a bit unusual as a language in that it neither runs in an interpreter (as Python and Ruby do) nor compiles down to a lower-level language (as Java and C do). Instead, it compiles to another high-level language, JavaScript. It is this JavaScript that runs, not your TypeScript. So TypeScript's relationship with JavaScript is essential, but it can also be a source of confusion. Understanding this relationship will help you be a more effective TypeScript developer.

TypeScript's type system also has some unusual aspects that you should be aware of. Later chapters cover the type system in much greater detail, but this one will alert you to some of the surprises that it has in store.

Item 1: Understand the Relationship Between TypeScript and JavaScript

If you use TypeScript for long, you'll inevitably hear the phrase "TypeScript is a superset of JavaScript" or "TypeScript is a typed superset of JavaScript." But what does this mean, exactly? And what is the relationship between TypeScript and JavaScript? Since these languages are so closely linked, a strong understanding of how they relate to each is the foundation for using TypeScript well.

TypeScript is a superset of JavaScript in a syntactic sense: so long as your JavaScript program doesn't have any syntax errors then it is also a TypeScript program. It's quite likely that TypeScript's type checker will flag some issues with your code. But this is an independent problem. TypeScript will still parse your code and emit JavaScript. (This is another key part of the relationship. We'll explore this more in Item 3.)

TypeScript files use a *.ts* (or *.tsx*) extension, rather than the *.js* (or *.jsx*) extension of a JavaScript file. This doesn't mean that TypeScript is a completely different language! Since TypeScript is a superset of JavaScript, the code in your *.js* files is already TypeScript. Renaming *main.js* to *main.ts* doesn't change that.

This is enormously helpful if you're migrating an existing JavaScript codebase to TypeScript. It means that you don't have to rewrite any of your code in another language to start using TypeScript and get the benefits it provides. This would not be true if you chose to rewrite your JavaScript in a language like Java. This gentle migration path is one of the best features of TypeScript. There will be much more to say about this topic in Chapter 8.

All JavaScript programs are TypeScript programs, but the converse is not true: there are TypeScript programs which are not JavaScript programs. This is because TypeScript adds additional syntax for specifying types. (There are some other bits of syntax it adds, largely for historical reasons. See Item 53.)

For instance, this is a valid TypeScript program:

```
function greet(who: string) {
  console.log('Hello', who);
}
```

But when you run this through a program like `node` that expects JavaScript, you'll get an error:

```
function greet(who: string) {
                  ^

SyntaxError: Unexpected token :
```

The `: string` is a type annotation that is specific to TypeScript. Once you use one, you've gone beyond plain JavaScript (see Figure 1-1).

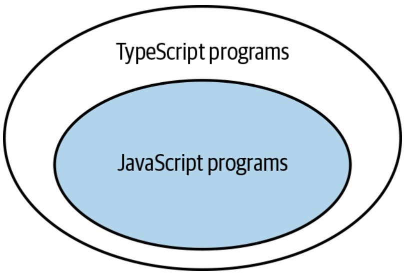


Figure 1-1. All JavaScript is TypeScript, but not all TypeScript is JavaScript

This is not to say that TypeScript doesn't provide value for plain JavaScript programs. It does! For example, this JavaScript program:

```
let city = 'new york city';
console.log(city.toUppercase());
```

will throw an error when you run it:

```
TypeError: city.toUppercase is not a function
```

There are no type annotations in this program, but TypeScript's type checker is still able to spot the problem:

```
let city = 'new york city';
console.log(city.toUppercase());
              // ~~~~~~~~~~~ Property 'toUppercase' does not exist on type
              //             'string'. Did you mean 'toUpperCase'?
```

You didn't have to tell TypeScript that the type of `city` was `string`: it inferred it from the initial value. Type inference is a key part of TypeScript and Chapter 3 explores how to use it well.

One of the goals of TypeScript's type system is to detect code that will throw an exception at runtime, without having to run your code. When you hear TypeScript described as a "static" type system, this is what it refers to. The type checker cannot always spot code that will throw exceptions, but it will try.

Even if your code doesn't throw an exception, it still might not do what you intend. TypeScript tries to catch some of these issues, too. For example, this JavaScript program:

```
const states = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capital: 'Juneau'},
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capitol);
}
```

will log:

```
undefined
undefined
undefined
```

Whoops! What went wrong? This program is valid JavaScript (and hence Type-Script). And it ran without throwing any errors. But it clearly didn't do what you intended. Even without adding type annotations, TypeScript's type checker is able to spot the error (and offer a helpful suggestion):

```
for (const state of states) {
  console.log(state.capitol);
                  // ~~~~~~~ Property 'capitol' does not exist on type
                  //         '{ name: string; capital: string; }'.
                  //         Did you mean 'capital'?
}
```

While TypeScript can catch errors even if you don't provide type annotations, it's able to do a much more thorough job if you do. This is because type annotations tell TypeScript what your *intent* is, and this lets it spot places where your code's behavior does not match your intent. For example, what if you'd reversed the `capital/capitol` spelling mistake in the previous example?

```
const states = [
  {name: 'Alabama', capitol: 'Montgomery'},
  {name: 'Alaska',  capitol: 'Juneau'},
  {name: 'Arizona', capitol: 'Phoenix'},
  // ...
];
for (const state of states) {
  console.log(state.capital);
                  // ~~~~~~~ Property 'capital' does not exist on type
                  //         '{ name: string; capitol: string; }'.
                  //         Did you mean 'capitol'?
}
```

The error that was so helpful before now gets it exactly wrong! The problem is that you've spelled the same property two different ways, and TypeScript doesn't know which one is right. It can guess, but it may not always be correct. The solution is to clarify your intent by explicitly declaring the type of `states`:

```
interface State {
  name: string;
  capital: string;
}
const states: State[] = [
  {name: 'Alabama', capitol: 'Montgomery'},
                  // ~~~~~~~~~~~~~~~~~~~~~
  {name: 'Alaska',  capitol: 'Juneau'},
                  // ~~~~~~~~~~~~~~~~~
  {name: 'Arizona', capitol: 'Phoenix'},
                  // ~~~~~~~~~~~~~~~~~~ Object literal may only specify known
                  //         properties, but 'capitol' does not exist in type
                  //         'State'.  Did you mean to write 'capital'?
  // ...
];
for (const state of states) {
  console.log(state.capital);
}
```

Now the errors match the problem and the suggested fix is correct. By spelling out our intent, you've also helped TypeScript spot other potential problems. For instance, had you only misspelled `capitol` once in the array, there wouldn't have been an error before. But with the type annotation, there is:

```
const states: State[] = [
  {name: 'Alabama', capital: 'Montgomery'},
  {name: 'Alaska',  capitol: 'Juneau'},
                 // ~~~~~~~~~~~~~~~~~ Did you mean to write 'capital'?
  {name: 'Arizona', capital: 'Phoenix'},
  // ...
];
```

In terms of the Venn diagram, we can add in a new group of programs: TypeScript programs which pass the type checker (see Figure 1-2).

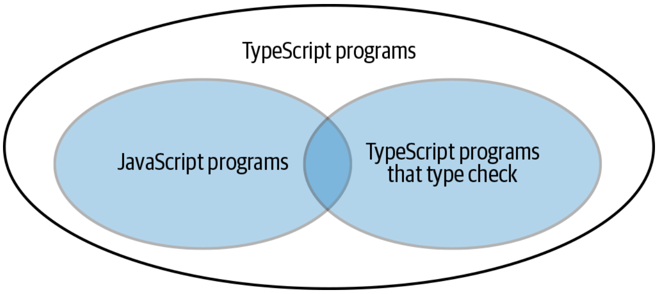


Figure 1-2. All JavaScript programs are TypeScript programs. But only some JavaScript (and TypeScript) programs pass the type checker.

If the statement that "TypeScript is a superset of JavaScript" feels wrong to you, it may be because you're thinking of this third set of programs in the diagram. In practice, this is the most relevant one to the day-to-day experience of using TypeScript. Generally when you use TypeScript, you try to keep your code passing all the type checks.

TypeScript's type system *models* the runtime behavior of JavaScript. This may result in some surprises if you're coming from a language with stricter runtime checks. For example:

```
const x = 2 + '3';  // OK, type is string
const y = '2' + 3;  // OK, type is string
```

These statements both pass the type checker, even though they are questionable and do produce runtime errors in many other languages. But this does model the runtime behavior of JavaScript, where both expressions result in the string `"23"`.

TypeScript does draw the line somewhere, though. The type checker flags issues in all of these statements, even though they do not throw exceptions at runtime:

```
const a = null + 7;  // Evaluates to 7 in JS
       // ~~~~ Operator '+' cannot be applied to types ...
const b = [] + 12;  // Evaluates to '12' in JS
```

```
        // ~~~~~~~ Operator '+' cannot be applied to types ...
alert('Hello', 'TypeScript');  // alerts "Hello"
             // ~~~~~~~~~~~~ Expected 0-1 arguments, but got 2
```

The guiding principle of TypeScript's type system is that it should model JavaScript's runtime behavior. But in all of these cases, TypeScript considers it more likely that the odd usage is the result of an error than the developer's intent, so it goes beyond simply modeling the runtime behavior. We saw another example of this in the `capital`/`capitol` example, where the program didn't throw (it logged `undefined`) but the type checker still flagged an error.

How does TypeScript decide when to model JavaScript's runtime behavior and when to go beyond it? Ultimately this is a matter of taste. By adopting TypeScript you're trusting the judgment of the team that builds it. If you enjoy adding `null` and `7` or `[]` and `12`, or calling functions with superfluous arguments, then TypeScript might not be for you!

If your program type checks, could it still throw an error at runtime? The answer is "yes." Here's an example:

```
const names = ['Alice', 'Bob'];
console.log(names[2].toUpperCase());
```

When you run this, it throws:

```
TypeError: Cannot read property 'toUpperCase' of undefined
```

TypeScript assumed the array access would be within bounds, but it was not. The result was an exception.

Uncaught errors also frequently come up when you use the `any` type, which we'll discuss in Item 5 and in more detail in Chapter 5.

The root cause of these exceptions is that TypeScript's understanding of a value's type and reality have diverged. A type system which can guarantee the accuracy of its static types is said to be *sound*. TypeScript's type system is very much not sound, nor was it ever intended to be. If soundness is important to you, you may want to look at other languages like Reason or Elm. While these do offer more guarantees of runtime safety, this comes at a cost: neither is a superset of JavaScript, so migration will be more complicated.

Things to Remember

- TypeScript is a superset of JavaScript. In other words, all JavaScript programs are already TypeScript programs. TypeScript has some syntax of its own, so TypeScript programs are not, in general, valid JavaScript programs.
- TypeScript adds a type system that models JavaScript's runtime behavior and tries to spot code which will throw exceptions at runtime. But you shouldn't expect it

to flag every exception. It is possible for code to pass the type checker but still throw at runtime.

- While TypeScript's type system largely models JavaScript behavior, there are some constructs that JavaScript allows but TypeScript chooses to bar, such as calling functions with the wrong number of arguments. This is largely a matter of taste.

Item 2: Know Which TypeScript Options You're Using

Does this code pass the type checker?

```
function add(a, b) {
  return a + b;
}
add(10, null);
```

Without knowing which options you're using, it's impossible to say! The TypeScript compiler has an enormous set of these, nearly 100 at the time of this writing.

They can be set via the command line:

```
$ tsc --noImplicitAny program.ts
```

or via a configuration file, *tsconfig.json*:

```
{
  "compilerOptions": {
    "noImplicitAny": true
  }
}
```

You should prefer the configuration file. It ensures that your coworkers and tools all know exactly how you plan to use TypeScript. You can create one by running `tsc --init`.

Many of TypeScript's configuration settings control where it looks for source files and what sort of output it generates. But a few control core aspects of the language itself. These are high-level design choices that most languages do not leave to their users. TypeScript can feel like a very different language depending on how it is configured. To use it effectively, you should understand the most important of these settings: `noImplicitAny` and `strictNullChecks`.

`noImplicitAny` controls whether variables must have known types. This code is valid when `noImplicitAny` is off:

```
function add(a, b) {
  return a + b;
}
```

If you mouse over the `add` symbol in your editor, it will reveal what TypeScript has inferred about the type of that function:

```
function add(a: any, b: any): any
```

The `any` types effectively disable the type checker for code involving these parameters. `any` is a useful tool, but it should be used with caution. For much more on `any`, see Item 5 and Chapter 3.

These are called *implicit* `anys` because you never wrote the word "any" but still wound up with dangerous `any` types. This becomes an error if you set the `noImplicitAny` option:

```
function add(a, b) {
          // ~    Parameter 'a' implicitly has an 'any' type
          //    ~ Parameter 'b' implicitly has an 'any' type
  return a + b;
}
```

These errors can be fixed by explicitly writing type declarations, either `: any` or a more specific type:

```
function add(a: number, b: number) {
  return a + b;
}
```

TypeScript is the most helpful when it has type information, so you should be sure to set `noImplicitAny` whenever possible. Once you grow accustomed to all variables having types, TypeScript without `noImplicitAny` feels almost like a different language.

For new projects, you should start with `noImplicitAny` on, so that you write the types as you write your code. This will help TypeScript spot problems, improve the readability of your code, and enhance your development experience (see Item 6). Leaving `noImplicitAny` off is only appropriate if you're transitioning a project from JavaScript to TypeScript (see Chapter 8).

`strictNullChecks` controls whether `null` and `undefined` are permissible values in every type.

This code is valid when `strictNullChecks` is off:

```
const x: number = null;  // OK, null is a valid number
```

but triggers an error when you turn `strictNullChecks` on:

```
const x: number = null;
//    ~ Type 'null' is not assignable to type 'number'
```

A similar error would have occurred had you used `undefined` instead of `null`.

If you mean to allow `null`, you can fix the error by making your intent explicit:

```
const x: number | null = null;
```

If you do not wish to permit `null`, you'll need to track down where it came from and add either a check or an assertion:

```
    const el = document.getElementById('status');
    el.textContent = 'Ready';
 // ~~ Object is possibly 'null'

    if (el) {
      el.textContent = 'Ready';  // OK, null has been excluded
    }
    el!.textContent = 'Ready';  // OK, we've asserted that el is non-null
```

`strictNullChecks` is tremendously helpful for catching errors involving `null` and `undefined` values, but it does increase the difficulty of using the language. If you're starting a new project, try setting `strictNullChecks`. But if you're new to the language or migrating a JavaScript codebase, you may elect to leave it off. You should certainly set `noImplicitAny` before you set `strictNullChecks`.

If you choose to work without `strictNullChecks`, keep an eye out for the dreaded "undefined is not an object" runtime error. Every one of these is a reminder that you should consider enabling stricter checking. Changing this setting will only get harder as your project grows, so try not to wait too long before enabling it.

There are many other settings that affect language semantics (e.g., `noImplicitThis` and `strictFunctionTypes`), but these are minor compared to `noImplicitAny` and `strictNullChecks`. To enable all of these checks, turn on the `strict` setting. TypeScript is able to catch the most errors with `strict`, so this is where you eventually want to wind up.

Know which options you're using! If a coworker shares a TypeScript example and you're unable to reproduce their errors, make sure your compiler options are the same.

Things to Remember

- The TypeScript compiler includes several settings which affect core aspects of the language.
- Configure TypeScript using *tsconfig.json* rather than command-line options.
- Turn on `noImplicitAny` unless you are transitioning a JavaScript project to TypeScript.
- Use `strictNullChecks` to prevent "undefined is not an object"-style runtime errors.

- Aim to enable `strict` to get the most thorough checking that TypeScript can offer.

Item 3: Understand That Code Generation Is Independent of Types

At a high level, `tsc` (the TypeScript compiler) does two things:

- It converts next-generation TypeScript/JavaScript to an older version of JavaScript that works in browsers ("transpiling").
- It checks your code for type errors.

What's surprising is that these two behaviors are entirely independent of one another. Put another way, the types in your code cannot affect the JavaScript that TypeScript emits. Since it's this JavaScript that gets executed, this means that your types can't affect the way your code runs.

This has some surprising implications and should inform your expectations about what TypeScript can and cannot do for you.

Code with Type Errors Can Produce Output

Because code output is independent of type checking, it follows that code with type errors can produce output!

```
$ cat test.ts
let x = 'hello';
x = 1234;
$ tsc test.ts
test.ts:2:1 - error TS2322: Type '1234' is not assignable to type 'string'

2 x = 1234;
  ~

$ cat test.js
var x = 'hello';
x = 1234;
```

This can be quite surprising if you're coming from a language like C or Java where type checking and output go hand in hand. You can think of all TypeScript errors as being similar to warnings in those languages: it's likely that they indicate a problem and are worth investigating, but they won't stop the build.

Compiling and Type Checking

This is likely the source of some sloppy language that is common around TypeScript. You'll often hear people say that their TypeScript "doesn't compile" as a way of saying that it has errors. But this isn't technically correct! Only the code generation is "compiling." So long as your TypeScript is valid JavaScript (and often even if it isn't), the TypeScript compiler will produce output. It's better to say that your code has errors, or that it "doesn't type check."

Code emission in the presence of errors is helpful in practice. If you're building a web application, you may know that there are problems with a particular part of it. But because TypeScript will still generate code in the presence of errors, you can test the other parts of your application before you fix them.

You should aim for zero errors when you commit code, lest you fall into the trap of having to remember what is an expected or unexpected error. If you want to disable output on errors, you can use the `noEmitOnError` option in *tsconfig.json*, or the equivalent in your build tool.

You Cannot Check TypeScript Types at Runtime

You may be tempted to write code like this:

```
interface Square {
  width: number;
}
interface Rectangle extends Square {
  height: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
                    // ~~~~~~~~~ 'Rectangle' only refers to a type,
                    //           but is being used as a value here
    return shape.width * shape.height;
                    //         ~~~~~~ Property 'height' does not exist
                    //                on type 'Shape'
  } else {
    return shape.width * shape.width;
  }
}
```

The `instanceof` check happens at runtime, but `Rectangle` is a type and so it cannot affect the runtime behavior of the code. TypeScript types are "erasable": part of compilation to JavaScript is simply removing all the `interface`s, `type`s, and type annotations from your code.

To ascertain the type of shape you're dealing with, you'll need some way to reconstruct its type at runtime. In this case you can check for the presence of a `height` property:

```
function calculateArea(shape: Shape) {
  if ('height' in shape) {
    shape;  // Type is Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Type is Square
    return shape.width * shape.width;
  }
}
```

This works because the property check only involves values available at runtime, but still allows the type checker to refine `shape`'s type to `Rectangle`.

Another way would have been to introduce a "tag" to explicitly store the type in a way that's available at runtime:

```
interface Square {
  kind: 'square';
  width: number;
}
interface Rectangle {
  kind: 'rectangle';
  height: number;
  width: number;
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape.kind === 'rectangle') {
    shape;  // Type is Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Type is Square
    return shape.width * shape.width;
  }
}
```

The `Shape` type here is an example of a "tagged union." Because they make it so easy to recover type information at runtime, tagged unions are ubiquitous in TypeScript.

Some constructs introduce both a type (which is not available at runtime) and a value (which is). The `class` keyword is one of these. Making `Square` and `Rectangle` classes would have been another way to fix the error:

```
class Square {
  constructor(public width: number) {}
}
class Rectangle extends Square {
  constructor(public width: number, public height: number) {
    super(width);
  }
}
type Shape = Square | Rectangle;

function calculateArea(shape: Shape) {
  if (shape instanceof Rectangle) {
    shape;  // Type is Rectangle
    return shape.width * shape.height;
  } else {
    shape;  // Type is Square
    return shape.width * shape.width;  // OK
  }
}
```

This works because `class Rectangle` introduces both a type and a value, whereas `interface` only introduced a type.

The `Rectangle` in `type Shape = Square | Rectangle` refers to the *type*, but the `Rectangle` in `shape instanceof Rectangle` refers to the *value*. This distinction is important to understand but can be quite subtle. See Item 8.

Type Operations Cannot Affect Runtime Values

Suppose you have a value that could be a string or a number and you'd like to normalize it so that it's always a number. Here's a misguided attempt that the type checker accepts:

```
function asNumber(val: number | string): number {
  return val as number;
}
```

Looking at the generated JavaScript makes it clear what this function really does:

```
function asNumber(val) {
  return val;
}
```

There is no conversion going on whatsoever. The `as number` is a type operation, so it cannot affect the runtime behavior of your code. To normalize the value you'll need to check its runtime type and do the conversion using JavaScript constructs:

```
function asNumber(val: number | string): number {
  return typeof(val) === 'string' ? Number(val) : val;
}
```

(`as number` is a *type assertion*. For more on when it's appropriate to use these, see Item 9.)

Runtime Types May Not Be the Same as Declared Types

Could this function ever hit the final `console.log`?

```
function setLightSwitch(value: boolean) {
  switch (value) {
    case true:
      turnLightOn();
      break;
    case false:
      turnLightOff();
      break;
    default:
      console.log(`I'm afraid I can't do that.`);
  }
}
```

TypeScript usually flags dead code, but it does not complain about this, even with the `strict` option. How could you hit this branch?

The key is to remember that `boolean` is the *declared* type. Because it is a TypeScript type, it goes away at runtime. In JavaScript code, a user might inadvertently call `setLightSwitch` with a value like `"ON"`.

There are ways to trigger this code path in pure TypeScript, too. Perhaps the function is called with a value which comes from a network call:

```
interface LightApiResponse {
  lightSwitchValue: boolean;
}
async function setLight() {
  const response = await fetch('/light');
  const result: LightApiResponse = await response.json();
  setLightSwitch(result.lightSwitchValue);
}
```

You've declared that the result of the `/light` request is `LightApiResponse`, but nothing enforces this. If you misunderstood the API and `lightSwitchValue` is really a `string`, then a string will be passed to `setLightSwitch` at runtime. Or perhaps the API changed after you deployed.

TypeScript can get quite confusing when your runtime types don't match the declared types, and this is a situation you should avoid whenever you can. But be aware that it's possible for a value to have types other than the ones you've declared.

You Cannot Overload a Function Based on TypeScript Types

Languages like C++ allow you to define multiple versions of a function that differ only in the types of their parameters. This is called "function overloading." Because the runtime behavior of your code is independent of its TypeScript types, this construct isn't possible in TypeScript:

```
function add(a: number, b: number) { return a + b; }
      // ~~~ Duplicate function implementation
function add(a: string, b: string) { return a + b; }
      // ~~~ Duplicate function implementation
```

TypeScript *does* provide a facility for overloading functions, but it operates entirely at the type level. You can provide multiple declarations for a function, but only a single implementation:

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;

function add(a, b) {
  return a + b;
}

const three = add(1, 2);  // Type is number
const twelve = add('1', '2');  // Type is string
```

The first two declarations of `add` only provide type information. When TypeScript produces JavaScript output, they are removed, and only the implementation remains. (If you use this style of overloading, take a look at Item 50 first. There are some subtleties to be aware of.)

TypeScript Types Have No Effect on Runtime Performance

Because types and type operations are erased when you generate JavaScript, they cannot have an effect on runtime performance. TypeScript's static types are truly zero cost. The next time someone offers runtime overhead as a reason to not use TypeScript, you'll know exactly how well they've tested this claim!

There are two caveats to this:

- While there is no *runtime* overhead, the TypeScript compiler will introduce *build time* overhead. The TypeScript team takes compiler performance seriously and compilation is usually quite fast, especially for incremental builds. If the overhead becomes significant, your build tool may have a "transpile only" option to skip the type checking.
- The code that TypeScript emits to support older runtimes *may* incur a performance overhead vs. native implementations. For example, if you use generator functions and target ES5, which predates generators, then `tsc` will emit some helper code to make things work. This may have some overhead vs. a native implementation of generators. In any case, this has to do with the emit target and language levels and is still independent of the *types*.

Things to Remember

- Code generation is independent of the type system. This means that TypeScript types cannot affect the runtime behavior or performance of your code.
- It is possible for a program with type errors to produce code ("compile").
- TypeScript types are not available at runtime. To query a type at runtime, you need some way to reconstruct it. Tagged unions and property checking are common ways to do this. Some constructs, such as `class`, introduce both a TypeScript type and a value that is available at runtime.

Item 4: Get Comfortable with Structural Typing

JavaScript is inherently duck typed: if you pass a function a value with all the right properties, it won't care how you made the value. It will just use it. ("If it walks like a duck and talks like a duck…") TypeScript models this behavior, and it can sometimes lead to surprising results because the type checker's understanding of a type may be broader than what you had in mind. Having a good grasp of structural typing will help you make sense of errors and non-errors and help you write more robust code.

Say you're working on a physics library and have a 2D vector type:

```
interface Vector2D {
  x: number;
  y: number;
}
```

You write a function to calculate its length:

```
function calculateLength(v: Vector2D) {
  return Math.sqrt(v.x * v.x + v.y * v.y);
}
```

Now you introduce the notion of a named vector:

```
interface NamedVector {
  name: string;
  x: number;
  y: number;
}
```

The `calculateLength` function will work with `NamedVector`s because they have `x` and `y` properties, which are `number`s. TypeScript is smart enough to figure this out:

```
const v: NamedVector = { x: 3, y: 4, name: 'Zee' };
calculateLength(v);  // OK, result is 5
```

What is interesting is that you never declared the relationship between `Vector2D` and `NamedVector`. And you did not have to write an alternative implementation of `calculateLength` calculateLength for `NamedVectors`. TypeScript's type system is modeling JavaScript's runtime behavior (Item 1). It allowed `calculateLength` to be called with a `NamedVector` because its *structure* was compatible with `Vector2D`. This is where the term "structural typing" comes from.

But this can also lead to trouble. Say you add a 3D vector type:

```
interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```

and write a function to normalize them (make their length 1):

```
function normalize(v: Vector3D) {
  const length = calculateLength(v);
  return {
    x: v.x / length,
    y: v.y / length,
    z: v.z / length,
  };
}
```

If you call this function, you're likely to get something longer than unit length:

```
> normalize({x: 3, y: 4, z: 5})
{ x: 0.6, y: 0.8, z: 1 }
```

So what went wrong and why didn't TypeScript catch the error?

The bug is that `calculateLength` operates on 2D vectors but `normalize` operates on 3D vectors. So the `z` component is ignored in the normalization.

What's perhaps more surprising is that the type checker does not catch this issue. Why are you allowed to call `calculateLength` with a 3D vector, despite its type declaration saying that it takes 2D vectors?

What worked so well with named vectors has backfired here. Calling `calculateLength` with an `{x, y, z}` object doesn't throw an error. So the type checker doesn't complain, either, and this behavior has led to a bug. (If you want this to be an error, you have some options. We'll return to this example in Item 37.)

As you write functions, it's easy to imagine that they will be called with arguments having the properties you've declared *and no others*. This is known as a "sealed" or "precise" type, and it cannot be expressed in TypeScript's type system. Like it or not, your types are "open."

This can sometimes lead to surprises:

```
function calculateLengthL1(v: Vector3D) {
  let length = 0;
  for (const axis of Object.keys(v)) {
    const coord = v[axis];
               // ~~~~~~~ Element implicitly has an 'any' type because ...
               //         'string' can't be used to index type 'Vector3D'
    length += Math.abs(coord);
  }
  return length;
}
```

Why is this an error? Since `axis` is one of the keys of `v`, which is a `Vector3D`, it should be either `"x"`, `"y"`, or `"z"`. And according to the declaration of `Vector3D`, these are all `number`s, so shouldn't the type of `coord` be `number`?

Is this error a false positive? No! TypeScript is correct to complain. The logic in the previous paragraph assumes that `Vector3D` is sealed and does not have other properties. But it could:

```
const vec3D = {x: 3, y: 4, z: 1, address: '123 Broadway'};
calculateLengthL1(vec3D);  // OK, returns NaN
```

Since `v` could conceivably have any properties, the type of `axis` is `string`. TypeScript has no reason to believe that `v[axis]` is a number because, as you just saw, it might not be. Iterating over objects can be tricky to type correctly. We'll return to this topic in Item 54, but in this case an implementation without loops would be better:

```
function calculateLengthL1(v: Vector3D) {
  return Math.abs(v.x) + Math.abs(v.y) + Math.abs(v.z);
}
```

Structural typing can also lead to surprises with `class`es, which are compared structurally for assignability:

```
class C {
  foo: string;
  constructor(foo: string) {
    this.foo = foo;
  }
}

const c = new C('instance of C');
const d: C = { foo: 'object literal' };  // OK!
```

Why is `d` assignable to `C`? It has a `foo` property that is a `string`. In addition, it has a `constructor` (from `Object.prototype`) that can be called with one argument (though it is usually called with zero). So the structures match. This might lead to surprises if you have logic in `C`'s constructor and write a function that assumes it's run. This is quite different from languages like C++ or Java, where declaring a parameter of type `C` guarantees that it will be either `C` or a subclass of it.

Structural typing is beneficial when you're writing tests. Say you have a function that runs a query on a database and processes the results:

```
interface Author {
  first: string;
  last: string;
}
function getAuthors(database: PostgresDB): Author[] {
  const authorRows = database.runQuery(`SELECT FIRST, LAST FROM AUTHORS`);
  return authorRows.map(row => ({first: row[0], last: row[1]}));
}
```

To test this, you could create a mock `PostgresDB`. But a better approach is to use structural typing and define a narrower interface:

```
interface DB {
  runQuery: (sql: string) => any[];
}
function getAuthors(database: DB): Author[] {
  const authorRows = database.runQuery(`SELECT FIRST, LAST FROM AUTHORS`);
  return authorRows.map(row => ({first: row[0], last: row[1]}));
}
```

You can still pass `getAuthors` a `PostgresDB` in production since it has a `runQuery` method. Because of structural typing, the `PostgresDB` doesn't need to say that it implements `DB`. TypeScript will figure out that it does.

When you write your tests, you can pass in a simpler object instead:

```
test('getAuthors', () => {
  const authors = getAuthors({
    runQuery(sql: string) {
      return [['Toni', 'Morrison'], ['Maya', 'Angelou']];
    }
  });
  expect(authors).toEqual([
    {first: 'Toni', last: 'Morrison'},
    {first: 'Maya', last: 'Angelou'}
  ]);
});
```

TypeScript will verify that our test `DB` conforms to the interface. And your tests don't need to know anything about your production database: no mocking libraries necessary! By introducing an abstraction (`DB`), we've freed our logic (and tests) from the details of a specific implementation (`PostgresDB`).

Another advantage of structural typing is that it can cleanly sever dependencies between libraries. For more on this, see Item 51.

Things to Remember

- Understand that JavaScript is duck typed and TypeScript uses structural typing to model this: values assignable to your interfaces might have properties beyond those explicitly listed in your type declarations. Types are not "sealed."

- Be aware that classes also follow structural typing rules. You may not have an instance of the class you expect!
- Use structural typing to facilitate unit testing.

Item 5: Limit Use of the any Type

TypeScript's type system is *gradual* and *optional*: *gradual* because you can add types to your code bit by bit and *optional* because you can disable the type checker whenever you like. The key to these features is the `any` type:

```
   let age: number;
   age = '12';
// ~~~ Type '"12"' is not assignable to type 'number'
   age = '12' as any;  // OK
```

The type checker is right to complain here, but you can silence it just by typing `as any`. As you start using TypeScript, it's tempting to use `any` types and type assertions (`as any`) when you don't understand an error, think the type checker is incorrect, or simply don't want to take the time to write out type declarations. In some cases this may be OK, but be aware that `any` eliminates many of the advantages of using TypeScript. You should at least understand its dangers before you use it.

There's No Type Safety with any Types

In the preceding example, the type declaration says that `age` is a `number`. But `any` lets you assign a `string` to it. The type checker will believe that it's a `number` (that's what you said, after all), and the chaos will go uncaught:

```
age += 1;  // OK; at runtime, age is now "121"
```

any Lets You Break Contracts

When you write a function, you are specifying a contract: if the caller gives you a certain type of input, you'll produce a certain type of output. But with an `any` type you can break these contracts:

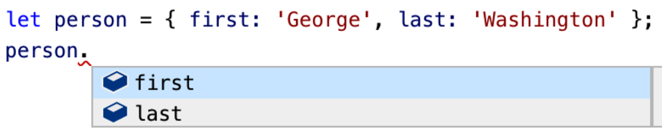
```
function calculateAge(birthDate: Date): number {
  // ...
}

let birthDate: any = '1990-01-19';
calculateAge(birthDate);  // OK
```

The birth date parameter should be a `Date`, not a `string`. The `any` type has let you break the contract of `calculateAge`. This can be particularly problematic because

JavaScript is often willing to implicitly convert between types. A `string` will sometimes work where a `number` is expected, only to break in other circumstances.

There Are No Language Services for any Types

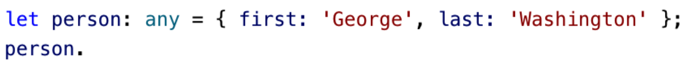
When a symbol has a type, the TypeScript language services are able to provide intelligent autocomplete and contextual documentation (as shown in Figure 1-3).



```
let person = { first: 'George', last: 'Washington' };
person.
```

Figure 1-3. The TypeScript Language Service is able to provide contextual autocomplete for symbols with types.

but for symbols with an `any` type, you're on your own (Figure 1-4).



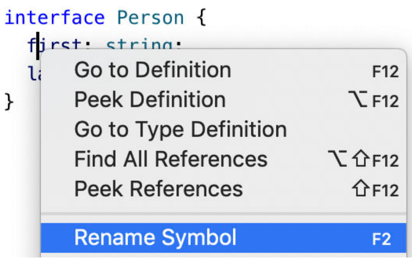
```
let person: any = { first: 'George', last: 'Washington' };
person.
```

Figure 1-4. There is no autocomplete for properties on symbols with any types.

Renaming is another such service. If you have a Person type and functions to format a person's name:

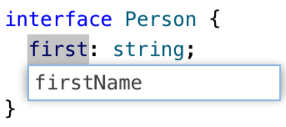
```
interface Person {
  first: string;
  last: string;
}

const formatName = (p: Person) => `${p.first} ${p.last}`;
const formatNameAny = (p: any) => `${p.first} ${p.last}`;
```

then you can select `first` in your editor, choose "Rename Symbol," and change it to `firstName` (see Figures 1-5 and 1-6).



```
interface Person {
  first: string;
  la
}
```

Figure 1-5. Renaming a symbol in vscode.



```
interface Person {
  first: string;
    firstName
}
```

Figure 1-6. Choosing the new name. The TypeScript language service ensures that all uses of the symbol in the project are also renamed.

This changes the `formatName` function but not the `any` version:

```
interface Person {
  firstName: string;
  last: string;
}
const formatName = (p: Person) => `${p.firstName} ${p.last}`;
const formatNameAny = (p: any) => `${p.first} ${p.last}`;
```

TypeScript's motto is "JavaScript that scales." A key part of "scales" is the language services, which are a core part of the TypeScript experience (see Item 6). Losing them will lead to a loss in productivity, not just for you but for everyone else working with your code.

any Types Mask Bugs When You Refactor Code

Suppose you're building a web application in which users can select some sort of item. One of your components might have an `onSelectItem` callback. Writing a type for an Item seems like a hassle, so you just use `any` as a stand-in:

```
interface ComponentProps {
  onSelectItem: (item: any) => void;
}
```

Here's code that manages that component:

```
function renderSelector(props: ComponentProps) { /* ... */ }

let selectedId: number = 0;
```

```
function handleSelectItem(item: any) {
  selectedId = item.id;
}

renderSelector({onSelectItem: handleSelectItem});
```

Later you rework the selector in a way that makes it harder to pass the whole `item` object through to `onSelectItem`. But that's no big deal since you just need the ID. You change the signature in `ComponentProps`:

```
interface ComponentProps {
  onSelectItem: (id: number) => void;
}
```

You update the component and everything passes the type checker. Victory!

…or is it? `handleSelectItem` takes an `any` parameter, so it's just as happy with an Item as it is with an ID. It produces a runtime exception, despite passing the type checker. Had you used a more specific type, this would have been caught by the type checker.

any Hides Your Type Design

The type definition for complex objects like your application state can get quite long. Rather than writing out types for the dozens of properties in your page's state, you may be tempted to just use an `any` type and be done with it.

This is problematic for all the reasons listed in this item. But it's also problematic because it hides the design of your state. As Chapter 4 explains, good type design is essential for writing clean, correct, and understandable code. With an `any` type, your type design is implicit. This makes it hard to know whether the design is a good one, or even what the design is at all. If you ask a coworker to review a change, they'll have to reconstruct whether and how you changed the application state. Better to write it out for everyone to see.

any Undermines Confidence in the Type System

Every time you make a mistake and the type checker catches it, it boosts your confidence in the type system. But when you see a type error at runtime, that confidence takes a hit. If you're introducing TypeScript on a larger team, this might make your coworkers question whether TypeScript is worth the effort. `any` types are often the source of these uncaught errors.

TypeScript aims to make your life easier, but TypeScript with lots of `any` types can be harder to work with than untyped JavaScript because you have to fix type errors *and* still keep track of the real types in your head. When your types match reality, it frees you from the burden of having to keep type information in your head. TypeScript will keep track of it for you.

For the times when you must use `any`, there are better and worse ways to do it. For much more on how to limit the downsides of `any`, see Chapter 5.

Things to Remember

- The `any` type effectively silences the type checker and TypeScript language services. It can mask real problems, harm developer experience, and undermine confidence in the type system. Avoid using it when you can!