Extracted from:

Modern Front-End Development for Rails, Second Edition

Hotwire, Stimulus, Turbo, and React

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Covers Rails 7

Modern Front-End Development for Rails

Second Edition

Hotwire, Stimulus, Turbo, and React

Noel Rappin

Edited by Katharine Dvorak

Modern Front-End Development for Rails, Second Edition

Hotwire, Stimulus, Turbo, and React

Noel Rappin

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-961-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2022

In programming, the *type* of a piece of variable data indicates the set of values that are valid for that data. For example, if I say that a variable is an integer, that means I expect the variable to have a value like `1`, `-3`, or `357` and not a value like `banana`. If the variable is set to a value that is not of the expected type, we think that something bad will happen. In some languages, the attempt to set data to the wrong value leads to a compiler error; in other languages it leads to incorrect or unspecified behavior at run time.

All high-level programing languages use some concept of type to determine behavior. Which is to say that all of these languages determine behavior by using not just the value of a variable in the system but also information about what kind of value it is.

TypeScript is a superset of JavaScript that optionally allows you to add annotations to the code to specify type information.[1] TypeScript requires a compilation step that enforces type consistency and converts valid TypeScript to JavaScript suitable for browsers. The type system that TypeScript uses makes inferences about types based on the code, even if you do not explicitly provide type information. The goal of using TypeScript is to reduce code errors, first by preventing type mismatches, and as you become more advanced, by making invalid states into compiler-time errors so they are impossible to achieve at run time. TypeScript only enforces type consistency at compile time; it provides no protection against things that might happen at run time.

In this chapter, we'll cover the basics of TypeScript's syntax and semantics and take a glimpse at more advanced features. Throughout the rest of the book, you'll learn about other TypeScript features as they become important in improving the code we will be writing. We've already used some TypeScript features in our concert app to make claims about data types. Now let's go a little deeper on the syntax and see what TypeScript makes possible.

Using TypeScript

We've already installed TypeScript back in Adding TypeScript, on page ? by using the tsc_watch tool to run TypeScript's type checker over our code base, and then using esbuild to convert our TypeScript code into JavaScript. TypeScript's configuration is managed by a `tsconfig.json` file, which basically specifies what TypeScript allows and the kind of code that TypeScript emits. (I'll talk about more about the `tsconfig.json` file in Chapter 13, Validating Code with Advanced TypeScript, on page ?.)

1. https://www.typescriptlang.org

With TypeScript in place and with tsc_watch, every time we make a file change the TypeScript type checker will run, and if all its type checking passes, esbuild will convert it to JavaScript that can be sent to the browser. If the type checking does not pass, the TypeScript compiler will return error messages explaining the problem.

The important bit here is that once the code is compiled, TypeScript is done. TypeScript provides no protection at run time if the behavior of your data does not match expectations. This is usually fine as long as all the data is created by your code, but it can be a problem if your code is accepting external data that has not been type checked (for example, incoming JSON data from a server). Client-side TypeScript can guarantee that you are dealing with the data consistently in your code, but it cannot guarantee that the incoming data has the structure you expect.

Understanding Basic TypeScript Types

At its most basic, TypeScript allows you to annotate any variable declaration with a type by using the syntax `: <type>`, as in `let x: number`. As we'll see, this can get more complicated, but the starting point is annotating variables with types.

TypeScript defines four basic types:

- `boolean`: A Boolean value must be either JavaScript's `true` or `false` value.
- `number`: JavaScript only has one numeric type for floating point numbers. TypeScript's number type supports floating point and integer literals, hex literals (`0xab32`), octal literals (`0o1234`), and binary literals (`0b10010`).
- `string`: TypeScript allows both single and double quotes as string delimiters and supports the backquote (\) syntax for template strings.
- `object`: TypeScript defines an object as anything that is a value and is not one of the previous types, so not just raw objects, but also all instances of classes. Normally, you'd use a more specific type annotation as described in the following, rather than using plain `object`.

Both `null` and `undefined` are also TypeScript types, and you can say something like `let z: null = null`, though it's not clear why you'd want to.

By default, TypeScript allows the values `null` and `undefined` to be assigned to any variable no matter what its declared type is. If you are familiar with other relatively modern static-typed languages like Elm, Rust, or Swift, you may know that those languages force you to explicitly declare when `null` is a valid value for a given variable. That is not the default case in TypeScript, presumably because forcing explicit null declarations would make dealing with existing external JavaScript libraries quite complicated. Also, allowing null and undefined values makes it easier to gradually add TypeScript to an existing codebase. However, there is a compiler option, --strictNullChecks, which prohibits assigning null to a value unless explicitly allowed.

Any type in TypeScript can be used as the basis of an array with two different syntaxes that work identically: string[] or Array<string>. The second syntax is an example of a more general TypeScript feature called *generic types*, which allows the same type structure—in this case an Array—to have a different internal type—in this case a string—while still retaining the same behavior no matter the internal type.

Generic types allow you to have type checking in cases where the fact that the type is consistent across a class or function is more important than what specific type is involved.

Data structures are a common use case for generic types. If you have a type that is a list, for example, and you want to write a method that returns the first element of the list, without generics, you might have to write that function signature differently for each potential type of data you might have in the list:

```
function getFirst(list: AStringList): string
function getFirst(list: ANumberList): number
function getFirst(list: AUserList): User
```

and so on. But there is a pattern here: the return value is always the same type as the values that make up the list.

TypeScript allows us to use generics to represent the pattern, like this:

```
function getFirst(list: AList<T>): T
```

The angle brackets here represent the generic type and the T is just an identifier and could be anything (or at least, anything starting with a capital letter). Single-letter identifiers are usually used, at least in part, to make a clear distinction between generics and specific types.

The elements of a TypeScript array need to all be of the same type. If for some reason you need a linear structure that has multiple types, first you should think really hard about whether that is what you really want (most likely you want a class instead). But if you do want something like that, TypeScript calls that a *tuple*, and the syntax looks like this:

```
let myTuple: [string, number, string] = ["Jennifer", 8, "Lee"]
```

If you access an element of the tuple within the declaration, in this case myTuple[0], myTuple[1], or myTuple[2], then the return value is inferred to be the type from that element's tuple declaration. So myTuple[0] is a string and so on. If for some reason you access an element with a higher index than the elements in the array, please don't do that. TypeScript will let you do this, and the inferred type of the return value is what TypeScript calls a *union type*, meaning that the value is a member of one or more basic types.

Static vs. Dynamic Typing

At the most abstract level, there are two different strategies for dealing with type information in a programing language: *static* or *dynamic*.

A language with static types requires that each variable be assigned a type when it is first declared. The language expects to know what the type of a value is and uses that information to constrain at compile time what values can be assigned to that variable.

Different static languages have different requirements for how types are assigned. Some languages, like Java, require the type of every variable to be explicitly stated when the variable is declared. Other languages, like TypeScript, allow for *type inference*. In TypeScript, if you assign a variable with a typical JavaScript assignment like this:

```
let x = "hello"
```

TypeScript infers from the assignment that x is meant to be a string, and does not require further information; you do not have to explicitly declare that x is a string. Later, if we try to say x = 3, the TypeScript compiler will flag this as an error because 3 is not a string.

Some static languages also infer that if there's a type like string, there is a type "array of string," whereas in others you need to explicitly define the existence of the array. Some languages require you to declare up front whether a value can be null; others don't.

On the other hand, a dynamically typed language, like Ruby or plain JavaScript, assigns types at run time. In a dynamic language, you do not need to ever declare the type of a variable in the code. The language checks type information at run time using the current value of a variable at the moment the language needs to determine behavior—for example, when a method is called on a variable.

Types still have meaning in a dynamic language even if the type is not explicitly assigned. In Ruby, a line of code like 2 + "3" will be an error, but the error will happen at run time rather than compile time. In most dynamic languages the code `x + y` will have different behavior if `x` and `y` are numbers than if they are strings, and this behavior will be determined based on the value of `x` each time the line of code is executed.

TypeScript turns JavaScript into a statically typed language. Whether or not this is a good change is a surprisingly hard question to answer empirically. Both the general case of static versus dynamic languages and the specific case of TypeScript versus JavaScript are debated endlessly, with actual data difficult to come by. Creating any kind of valid, reproducible, scientific evidence about the general usefulness of programming languages is challenging.

There are a few points that are not disputed…much:

- A static language will catch errors in compilation that would otherwise potentially remain uncaught.
- Static languages generally are more verbose than dynamic languages, and there is sometimes a time cost to getting the compiler to agree that what you want to do is valid. More modern static languages use type inference to minimize the extra verbosity.
- Dynamic languages are generally more flexible and are usually considered easier to write code in, at least for small programs.
- Static languages provide more information to code analysis tools, so editor and tool support is easier to create. They also provide more meta-information in general, which can be valuable as communication about the code on larger teams.

The idea is that in a good static typing system, the benefits of tool support, communication, and error prevention will outweigh the costs of yelling at the compiler trying to get it to let you do what you want. In the general case of static versus dynamic languages, I think there's a lot of room for debate. In the specific case of TypeScript versus JavaScript, I think there is good reason to think there's some benefit.