Typescript Mini Reference 2023

A Quick Guide to Typescript Programming Language

Harry Yoon

Version 1.0.8, 2023-05-14

Copyright

Typescript Mini Reference:

A Quick Guide to Typescript Programming Language

Published: January 2023

Harry Yoon
San Diego, California

ISBN: 9798374743494

Preface

Typescript is a high-level programming language that is *different* and *distinct* from Javascript. But, Typescript mostly uses the same Javascript (or, more precisely ECMAScript) syntax. This is a curse and blessing.

On the one hand, many Javascript developers find Typescript more approachable/accessible and easier to get started with. This is probably one of the reasons why Typescript has been so successful, e.g., in terms of its wide adoption.

On the other hand, few Javascript developers use Typescript to its full potential. For many, Typescript is just Javascript with simple type annotations. Once you you learn the fundamentals of Typescript, however, that cannot be further from the truth. Typescript is, deep down, a rather different language from Javascript, in many respects.

Typescript's marketing slogan, in the early days, used to be *TypeScript is a superset of JavaScript*. This phrase, when interpreted literally, does not mean very much. For one thing, a programming language is not a mathematical set, and hence one programming language cannot be a superset of another. Regardless, Typescript uses the same or similar Javascript syntax in many parts of its grammar, including (almost) all statements and expressions.

Typescript's extension over Javascript is primarily limited to types. Javascript is a dynamically and loosely typed language. It has pros and cons. For small projects, or for quick prototyping, dynamic languages like Javascript or Python can be extremely convenient. On the other hand, when you work on bigger and longer-term projects, using statically typed languages tends to be increasingly more advantageous.

Typescript's new slogan is *TypeScript is JavaScript with syntax for types*. And, it emphasizes the tooling aspect of the programming language, *at any scale*. Typescript is widely used with many Javascript application

frameworks such as Angular, React, and Vuejs, which are primarily intended for building large-scale Javascript apps. In fact, Typescript got a big break, as a new language, when the Angular team adopted Typescript as their default programming language for Angular version 2.0 (and, onward). As the saying goes, the rest is history. As of this writing (January 2023), React Native, another Javascript-based hybrid mobile app development framework, also adopted Typescript as their primary language.

This book is an *unofficial* Typescript language reference. Regardless of your background, and your experience with Javascript and other programming languages, you will learn the essence of Typescript, and the core programming concepts in Typescript.

This book mostly focuses on the language features that are related to static typing. If you have some familiarity with programming in Javascript, you can read the book more or less from beginning to end, and you will get the full picture of Typescript's (rather unique) type system, among other things. On the flip side, written as an informal reference, this book may not be ideal for complete beginners. This book is definitely not a tutorial on Typescript.

One thing to note is that although the official names of these two programming languages are JavaScript and TypeScript, we mainly use "simpler" names such as *Javascript* and *Typescript* in this book, and even refer to them just as JS or TS, for brevity.

As stated, this book is not an authoritative language reference. Although we have taken every effort to ensure the accuracy of the content, there still may be some errors or misrepresentations. The readers are encouraged to consult the official documentation while reading this book.

Dear Readers:

Please read b4 you purchase, or start investing your time on, this book.

A programming language is like a set of standard lego blocks. There are small ones and there are big ones. Some blocks are straight and some are L-shaped. You use these lego blocks to build spaceships or submarines or amusement parks. Likewise, you build programs by assembling these building blocks of a given programming language.

This book is a *language reference*, written in an informal style. It goes through each of these lego blocks, if you will. This book, however, does not teach you how to build a space shuttle or a sail boat. If this distinction is not clear to you, it's unlikely that you will benefit much from this book. This kind of language reference books that go through the syntax and semantics of the programming language broadly, but not necessarily in gory details, can be rather useful to programmers with a wide range of background and across different skill levels.

This book is not for complete beginners, however. When you start learning a foreign language, for instance, you do not start from the grammar. Likewise, this book will not be very useful to people who have little experience in real programming. On the other hand, if you have some experience programming in other languages, and if you want to quickly learn the essential elements of this particular language, then this book can suit your needs rather well.

Ultimately, only you can decide whether this book will be useful for you. But, as stated, this book is written for a wide audience, from beginner to intermediate. Even experienced programmers can benefit, e.g., by quickly going through books like this once in a while. We all tend to forget things, and a quick regular refresher is always a good idea. You will learn, or re-learn, something "new" every time.

Good luck!

Table of Contents

Copyright . . . 1
Preface . . . 2
1. Introduction . . . 9
2. Typescript Basics . . . 13
    2.1. What is Typescript? . . . 13
    2.2. Static Typing . . . 16
    2.3. Notes on Development Process . . . 18
    2.4. The Typescript Compiler . . . 20
    2.5. Typescript JSON Configuration File . . . 22
3. Module System . . . 29
    3.1. ES Modules . . . 29
    3.2. Typescript Modules . . . 29
    3.3. Module Exports . . . 30
    3.4. Module Imports . . . 33
    3.5. Typescript Namespaces . . . 35
4. Variables . . . 36
    4.1. The `const` Declaration . . . 36
    4.2. The `let` Declaration . . . 37
    4.3. The `var` Declaration . . . 38
5. Basic Types . . . 40
    5.1. Javascript Types . . . 40
    5.2. Strict Equality . . . 42
    5.3. Primitive Types . . . 44
    5.4. Literal Types . . . 45
    5.5. The `any` Type . . . 46
    5.6. The `unknown` Type . . . 47
    5.7. The `null` and `undefined` Types . . . 49

5.8. The `never` Type . . . 51
5.9. The `void` Type . . . 51
5.10. The `object` Type . . . 52
5.11. Function Types . . . 53
5.12. Array Types . . . 54
5.13. Tuple Types . . . 55
5.14. Enum Types . . . 57
6. Type Aliases . . . 58
7. Type Annotations . . . 60
7.1. Variable Annotations . . . 60
7.2. Function Annotations . . . 61
8. Assertions . . . 64
8.1. Type Assertions . . . 64
8.2. Const Assertions . . . 65
8.3. Non-Null Assertions . . . 67
9. Generics . . . 68
9.1. Why Generics? . . . 68
9.2. Generic Functions . . . 71
9.3. Generic Types . . . 73
9.4. Generic Type Constraints . . . 74
10. Arrays . . . 76
10.1. Generic `Array<T>` . . . 76
10.2. Generic `ReadonlyArray<T>` . . . 79
11. Algebraic Data Types . . . 80
11.1. Tuples . . . 80
11.2. Readonly Tuples . . . 85
11.3. Union Types . . . 85
11.4. Discriminated Unions . . . 88
11.5. Intersection Types . . . 89

12. Function Types . . . 91
12.1. Function Definitions . . . 91
12.2. Arrow Function Definitions . . . 92
12.3. Function Types . . . 93
12.4. Parameter List . . . 96
12.5. Optional Parameters . . . 98
12.6. The Rest Parameter . . . 99
12.7. Parameter Destructuring . . . 102
12.8. The `this` Parameter . . . 104
12.9. Typescript Function Overloading . . . 106
13. Object Types . . . 109
13.1. Object Literal Types . . . 109
13.2. Object Type Members . . . 111
13.3. Index Signatures . . . 114
13.4. Getters and Setters . . . 117
13.5. Member Methods . . . 120
13.6. Structural Subtyping . . . 121
14. Interfaces . . . 123
14.1. Interface Types . . . 123
14.2. Extending Interfaces . . . 124
15. Classes . . . 126
15.1. The ECMAScript Class . . . 126
15.2. The Typescript Class . . . 130
15.3. Abstract Classes . . . 133
15.4. Implementing Interfaces . . . 135
15.5. Generic Classes . . . 138
16. Type Narrowing . . . 140
16.1. Control Flow Analysis . . . 140
16.2. The `typeof` Type Guard . . . 140

16.3. The `instanceof` Type Guard . . . 141
16.4. The `in` Operator Narrowing . . . 142
16.5. Discriminated Unions . . . 143
17. Advanced Types . . . 147
17.1. Template Literal Types . . . 147
17.2. The `typeof` Type Operator . . . 148
17.3. The `keyof` Type Operator . . . 149
17.4. Indexed Access Types . . . 150
17.5. Conditional Types . . . 150
17.6. Mapped Types . . . 152
A. How to Use This Book . . . 153
Index . . . 155
About the Author . . . 174
About the Series . . . 175
Community Support . . . 176

Chapter 1. Introduction

A type determines what kind of values are valid for a given object and what sort of operations are available for the object, among other things. Types are the core part of all high-level programming languages. The main difference between the dynamically typed languages like Javascript and the statically typed languages like Typescript is when the types are checked, e.g., run time vs build time.

Programs written in statically typed languages are generally easier to read, whereas it tends to be easier to write programs in dynamically typed languages. Although this is a gross oversimplification, it generally holds true. Hence, statically typed languages are often preferred in a large project involving multiple developers. On the flip side, there is clearly an overhead in using statically typed languages.

That's precisely the difference between Typescript and Javascript. When you start using Typescript, there is an upfront cost, in term of learning curve and general setup, etc., but as the project grows bigger, the benefits quickly start to outweigh the additional cost.

We start the book with the absolute basics of Typescript, and the general development process using Typescript. We go through some basic usages of the Typescript compiler, *tsc*, and the configuration file, *tsconfig.json*. If you have used Typescript before, you can skip most of this first chapter.

Next, we discuss the high-level organization of Javascript and Typescript programs using ES modules. Except for truly trivial code, most Typescript programs will need to be organized into modules.

There are primarily two contexts in which typing is important. The types of variables and functions. Function parameters and return values are also variables, broadly speaking, and hence we primarily deal with the types of variables when programming in statically typed

languages like Typescript. In contrast, in languages like Javascript, variables do not have associated types. There are three different ways to declare variables in Javascript (and hence in Typescript). We discuss each of them in the Variable Declaration chapter.

As in any statically typed programming languages, the types of Typescript can be divided into simple types and compound types, which are built from other types. In the Basic Types chapter, we go through various simple types of Typescript, including Javascript primitive types like `number`, `string`, and `boolean` as well as `null` and `undefined`, and other special types like `any` and `unknown`. In addition, Typescript allows using simple values as types, called the literal types. We also briefly look at some of the most fundamental (non-simple) types in Typescript, such as arrays, tuples, enums, and functions.

In Typescript, we can define new custom types in various different ways. Some of them are syntactically defined without names (e.g., union types), and we can use type aliases to give them reusable names, if need be. As a matter of fact, it is rather convenient, although not required, to use type aliases for any types that are to be used more than once in a program.

In many statically typed programming languages, variables are *declared* with types. In Typescript, we *annotate* variables. Although it is not a precise distinction, type annotations are only relevant at build time, and they have no runtime implications. In the Type Annotations chapter, we describe how to annotate variables and functions. Type annotations and type inferences are discussed throughout the book, not just in this chapter.

Typescript's static type checker relies on the type annotations (and, type inference rules) to enforce type safety. In certain situations, the developer may have certain information that is not readily available to the type checker. In such a case, he or she can use various forms of assertions to convey that information to the Typescript compiler.

Dynamically and weakly typed languages like Javascript do not, and need not, have generics, which is sometimes called the parametrized types. Generics permits defining a series of related types using type parameters, each of which is still strongly typed. If you are coming from a purely Javascript background, generics may seem a bit strange. Therefore, we start the Generics chapter with a simple example that motivates the use of generics.

In any programming languages, array is one of the most basic and most fundamental data types. In Typescript, array is a generic type, and it has two variants, the (regular) array that corresponds to Javascript array, and the readonly array.

In the Algebraic Data Types chapter, we describe a few simple ways to create a new type from other types. Two of the most common such methods are union types and tuple types.

In the next chapter, Function Types, we go through some more details of the function types. The modern Javascript gives a lot of flexibility in defining and using functions. Typescript includes, for example, support for annotating various kinds of function parameters. In addition, functions can be "overloaded" in Typescript.

Everything is an object in Javascript. An object type in Typescript can be declared using the object literal type syntax or the interface syntax. There are some minor differences in terms of their syntax and what not, but they serve fundamentally the same purposes, that is, giving a Javascript object, or a variable, a specific type. It is largely a matter of preference to choose one over the other, except for special cases. For example, only interfaces can be *implemented* by classes. We go through object literal types and interfaces in two separate chapters. But, their common elements are distributed over the two chapters rather than to be repeated.

In general, object literal types and interfaces play more fundamental roles in Typescript than classes since Javascript is an object-based

programming language at its core. However, the modern Javascript's `class` provides a convenient way to create objects that are related to each other through their prototype chains. When a type hierarchy is important among a set of related types, class provides a way to support OOP-like type inheritance in Typescript, which corresponds to the prototype chain in Javascript at run time.

Although `class` was introduced to Javascript some years ago (ES2015), it may still be considered a foreign part of Javascript by some developers. We go through a few essential elements of ECMAScript class in the beginning of the Classes chapter. Then we go through some of the Typescript extensions to the ECMAScript class next.

The ultimate goal of Typescript is to reduce the chances of runtime errors in Javascript programs, among other things. The Typescript compiler does what is called the control flow analysis to understand, to a certain extent, the runtime behavior of the code. Through control flow analysis, Typescript can provide services, at compile time, that go beyond what is explicitly annotated. This is called the type narrowing.

In the final chapter of the book, Advanced Types, we discuss a few other ways to create a type from other types. In particular, we discuss conditional types and mapped types in the final sections.

Note that this book illustrates Typescript grammar primarily through examples, say, rather than using more formal (and hence more precise) notations. When an example does not make sense to you, we recommend that you try the example yourself. By running the example code, you may be able to get a better insight.

As a general convention, all examples are Typescript code by default. Javascript code uses a special notation. They are always written in REPL, e.g., a Web browser developer console. In addition, Javascript code in these examples does not use semicolons, as a simple visual aid to help distinguish Javascript and Typescript code more easily. Semicolons are optional both in Javascript and Typescript.

Chapter 2. Typescript Basics

2.1. What is Typescript?

Typescript is a statically typed general-purpose programming language. Unlike many other programming languages like Javascript or Python, however, you cannot directly run Typescript programs.

You compile (or, transpile) a Typescript program to a Javascript program first, and you run the generated Javascript program on a Javascript runtime such as Web browsers or Node.js. There are many programming languages that can generate Javascript code. So, why is Typescript special?

Typescript has a lot of syntactic similarities with Javascript. Except for static typing (e.g., type annotations), simple Typescript programs *look* almost identical to those of Javascript. There are some additional features beyond typing in Typescript, but nonetheless Typescript is based on Javascript.

This makes Typescript more accessible to the programmers who are familiar with Javascript. Clearly, this is by design. But, on the flip side, it is also incidental. There is no intrinsic reason why a language that generates Javascript code has to be just like Javascript. At least in theory, a program written in any programming language can be converted to a Javascript program. (And, many modern programming languages can produce Javascript code as their output.)

Typescript is a superset of Javascript, not just in terms of syntax but in terms of development process. Typescript feels like such a natural extension to Javascript-based dev workflow, and that is what makes Typescript so special. Many people, especially many Javascript developers, pick up Typescript, among many other languages that can help build "better" Javascript programs, primarily for this reason.

Here's a bird's eye view of the Typescript development process:

1. Write a Typescript program,
2. Transpile it to a Javascript program, and then
3. Deploy the generated Javascript program.

In general, types determine, among other things,

- What kind of values are allowed for a given variable, and
- What kind of operations can be performed on those variables and values.

In case of dynamically typed languages like Javascript, types are associated with values, or objects, and they are checked and verified *at run time*. The variables in those languages do not have types.

```
> let sum = 1 + 2                          ①
> console.log(`type = ${typeof(sum)}`)     ②
type = number
> sum = "hello " + "world"                 ③
> console.log(`type = ${typeof(sum)}`)     ④
type = string
```

① The type of the right-hand side expression `1 + 2` is *number*. The variable `sum` points to this value. As indicated earlier, we use this REPL programming style to indicate that it is a Javascript program.

② The `typeof(sum)` returns the type of the value this variable points to at run time, which is *number*.

③ The type of the right-hand side expression `"hello " + "world"` in this case is *string*. The variable `sum` now points to this new value.

④ This will print out *string* since `sum` currently refers to the object `"hello world"`, whose type is *string*.

In the statically typed languages like Typescript, on the other hand, all variables are associated with types as well. And, the compiler uses this information to check and verify the correctness of a program (as far as types are concerned), e.g., without having to wait until the program runs (and, possibly crashes, or *even worse*, silently produces an incorrect result). This is known as the static type checking.

```
let sum = 1 + 2;                          ①
// sum = "hello " + "world";              ②
```

① The type of the variable `sum` will be inferred to be a *number* since it is initialized with an expression whose type is `number`. According to our convention, this is a Typescript program.

② Attempting to assign a value of *string* will cause a *compile time* error. (Technically, we are transpiling a Typescript program to Javascript program, but the word "compilation" is more commonly and broadly used, in general.)

As alluded in the introduction, statically typed languages are not inherently "better", or "more correct", than dynamically typed languages. In fact, there is really nothing wrong with the JavaScript program shown above. It may be even puzzling to some developers why the Typescript compiler does not allow us to do such a simple and trivial thing in this sample code. It's a tradeoff. By using static typing, and adopting an arguably more restrictive programming style, we can potentially prevent a lot of runtime errors, thanks to the static tooling.

One of the main differences between Typescript and some other statically typed languages is that Typescript completely removes the TS-specific static type information during the transpilation. This is called the *type erasure*. This is natural because Javascript does not support such type information. At run time, the type information is limited to what Javascript provides.

2.2. Static Typing

2.2.1. Type annotations

One of the major differences between Typescript and Javascript programs is that Typescript programs include type annotations. In Typescript, you annotate variables, including function parameters and return values, with particular types. Syntactically, you specify the type after the variable, preceded by a colon `:`. For example,

```
1 let pop: number = 10_000_000_000;        ①
```

① The variable `pop` is declared to be the `number` type, and its initial value is consistent with the specified type.

For functions,

```
3 function add(a: number,
4              b: number): number {        ①
5     return a + b;
6 }
```

① We annotate the function parameters and the return value, if any. In this example, both parameters, `a` and `b`, as well as the return value, are numbers.

2.2.2. Static type checking

Type annotations can be useful to other programmers who read the program. They often serve as, or even supplant, code documentation. But, primarily, the type annotations are used by the compiler, and other developer tools like IDEs, to enforce certain rules so that the correct functioning of the program can be ensured, etc.

We just declared a variable `pop` and a function `add` with specific types above. If we try to use these variable and function "incorrectly", that is, in the way that is inconsistent with the declared types, the compiler will stop us from doing that.

In strongly typed languages, the very act of assigning a value of a different type to a variable can be a fatal error, e.g., because the memory layouts of values of different types can be fundamentally different. For Typescript, which is transpiled to the loosely typed language, Javascript, that is typically not the case. But, this kind of mixup can still lead to serious errors down the line, for example, by attempting to call a method on an object of a type that does not support that particular method. Hence, the Typescript compiler prevents us from doing that out of abundance of caution.

If we try to use the above variable and function incorrectly, for instance, as follows:

```
8 pop = "hello";                                    ①
9 add("a", true);
```

① The line numbers indicates that all these three code snippets are from a single script.

We will get the following errors when we try to compile the code,

```
index.ts:8:1 - error TS2322: Type 'string' is not assignable
to type 'number'.
8 pop = "hello";
  ~~~
index.ts:9:5 - error TS2345: Argument of type 'string' is not
assignable to parameter of type 'number'.
9 add("a", true);
      ~~~
Found 2 errors in the same file, starting at: index.ts:8
```

In many cases, you do not even have to compile the code to see the errors since the IDEs that support Typescript will let you know these errors immediately, through static analysis, while you are programming. IDEs often use what is called *intellisense* as well to help developers avoid these kinds of errors in the first place.

2.2.3. Type inference

In many cases, especially for local variable declarations, the type of the initial value of a variable may likely be the one that is intended for that variable. In such a case, its type may not have to be explicitly specified, and the TS compiler can infer the type.

```
let billion = 1_000_000_000;              ①
```

① The variable `billion` is declared with an initial `number` value, and hence if it is not explicitly annotated, its type is inferred to be `number`.

2.3. Notes on Development Process

Dynamic languages like Python and Javascript tend to provide faster iteration cycles for development. You can make a quick change to the code and run it again to see the result of the change. And, you can repeat these steps every time you make changes.

The development process with compiled languages tends go a little bit slower. There is an extra step of compilation at each cycle, which can hinder rapid iteration (relatively speaking). Furthermore, Typescript requires some upfront investment in terms of project setup, etc.

Hence, generally speaking, Typescript is not for small projects. If you are writing a simple few line program that is going to be included in a script tag of an HTML file, for instance, Typescript is clearly an overkill.

In fact, Typescript rarely provides any benefit when you are writing a small program because it is often a lot easier to spot some obvious errors such as mismatched types just by looking at the code rather than relying on tools.

As we move along the axis of *scale,* however, it gets more and more beneficial for bigger and longer-term projects to use Typescript. At some point, the overhead and benefit of using Typescript become reversed.

One thing to note is that many modern Javascript projects go through some kind of "build" steps before deployment. For example, tools like *Babel* are often used for transpiling Javascript code to a specific target platform, and tools like *Webpack* are often used for "bundling", which often includes the Babel transpilation as one of its build steps. Some projects may require minification and obfuscation of JS code, and so forth.

It is not uncommon to incorporate those steps into the development process. Hence, it is not totally out of place to include the Typescript transpilation step into these overall development process. In fact, many popular Javascript frameworks such as React, Angular, and Vuejs support Typescript as part of their native development workflow. Hence, the cost of incorporating TS into the overall development and/or deployment process is relatively small, if not completely zero, especially for large projects.

One additional overhead of using Typescript, if you will, is that the vast majority of Javascript libraries are (still) primarily written in Javascript, and they do not natively include type declarations. This is becoming less of an issue for more widely-used Javascript libraries now that Typescript is being widely adopted, but if you are interested in using smaller, less used, libraries, then you may occasionally run into some problems. You may have to go through some additional steps in order to be able to use Javascript libraries without type definition files.

This book primarily focuses on the Typescript language, and we will not discuss all the details and nuances surrounding TS-based app development process, but they are easy to pick up through experience and some trials and errors. (For instance, as a trivial example, if you want to use Typescript with Node.js, you will need to import Node's type definition file, i.e., *@types/node*, into your project as a dev dependency.)

2.4. The Typescript Compiler

You can install the official Typescript package via *npm* (or, other package managers such as *yarn* or *NuGet*), which includes Typescript core libraries as well as the build tools such as the Typescript compiler, *tsc*. For example,

```
$ npm i -g typescript@latest              ①
```

① Or, you can simply add the typescript package as a dev dependency to your project.

Let's try the *tsc* command:

```
$ tsc --version
Version 4.9.4                             ①
```

① Version 4.9.4 is the most recent version of Typescript, as of this writing.

The official Typescript package does not include a Typescript REPL, and there does not appear to be any widely-used implementation of REPL where you can evaluate Typescript expressions and interactively run Typescript statements. If you want to quickly test a simple TS code, you can use the TypeScript Playground [https://www.typescriptlang.org/play].

Typescript supports quite a few compiler settings that can be used to customize the behavior of the compiler, as well as the semantics of the language itself. Hence, it is rather common, and almost required, to use the Typescript JSON configuration file, *tsconfig.json*, along with *tsc*.

You can generate a default *tsconfig.json* file as follows:

```
$ tsc --init

Created a new tsconfig.json with:
  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true

$ ls
tsconfig.json
```

This output shows a few important settings from the *tsconfig.json* file. We will look at some of these settings in the next section.

You can compile just one or a few Typescript files. For example,

```
$ tsc hello.ts world.ts
```

It is, however, a more common practice to organize one or more Typescript files into a "project", and compile all, or most, of those file in the project together. The location of the *tsconfig.json* file is the root of a Typescript project. (In other words, *tsc --init* creates a new (implicit) project in the current folder, which should generally coincide with the root of other project-like structures, e.g., an *npm* project.)

For example, you can compile the project as follows:

```
$ ls
hello.ts  tsconfig.json  world.ts
$ tsc -p .
$ ls
hello.js  hello.ts  tsconfig.json  world.js  world.ts
```

The `-p`, or `--project`, flag is followed by the location of the *tsconfig.json* file, e.g., the current folder `.` in this example, which in turn designates the files to compile, either explicitly or implicitly. Note that `tsc` generated two JS files corresponding to the two TS files, `hello.ts` and `world.ts`. (Typescript, by default, treats files with extensions *.ts*, *.tsx*, and *.d.ts* as TS files.)

Furthermore, *tsc* includes many additional options. You can try *tsc --help* to view the basic usage of the compiler, or *tsc --all* to list all available options. One useful option during development is the *-w* (or *--watch*) flag, which starts the *tsc* command in the "watch mode". When Typescript source files in a given project are modified, it automatically compiles the code so that the generated JS files remain in sync with the TS files (e.g., with some build time lag).

2.5. Typescript JSON Configuration File

Javascript, or more precisely the ECMAScript language, has two *variants*. The language used with the strict mode, `"use strict"`, and the language used without that designation. Although the difference is relatively small, nonetheless they are two separate and incompatible languages.

Typescript has dozens of different variants, if not hundreds. Depending on the compiler settings, the language behaves differently. In theory, we have one variant of Typescript for each combination of Typescript compiler settings, and there are many such settings that affect the language behavior.

The default *tsconfig.json* file generated through `tsc --init`, for instance, includes many of these options, with most of them commented out, so that they can be easily tweaked if need be.

2.5.1. Top-level options

`include` and `exclude`

The top-level `include` option specifies which Typescript source files are part of a program (e.g., to be compiled by *tsc*). Its value is an array of file names or glob patterns, and they are resolved relative to the Typescript project root, e.g., the directory containing the *tsconfig.json* file. For example,

```
{
  "compilerOptions": {},
  "include": ["src/**/*", "tests/**/*"]
}
```

The `exclude` option can be used to specify the files that should not be part of the program from the `include` list. Its value is also an array of file names or glob patterns. Both `include` and `exclude` support the following wildcard characters for glob:

- `*` matches zero or more characters.
- `?` matches any one character.
- `**/` matches any directory nested to any level.

`files`

In some cases, it may be convenient to explicitly list all source files (e.g., TS or JS files) that make up a program. The `files` top-level option can be used for that purpose. For example,

```
{
  "files": [
    "hello-pele.ts",
    "hello-messi.ts",
    "hello-ronaldo.ts"
  ]
}
```

2.5.2. `compilerOptions`

All of the `compilerOptions` options can be important in certain contexts, but here are a few of the more significant settings with broader implications, in general.

`target`

The `target` value sets the target ECMAScript language version for emitted Javascript code, and it adds any necessary polyfills. Usually, the default value of `"es2016"` should be good enough.

```
{
  "compilerOptions": {
    "target": "es2016"
  }
}
```

(Note that you are not programming against this target. You program in Typescript, which is usually in line with the most recent version of ECMAScript. The `target` option determines the transpiled output, which should be somewhat conservatively set for compatibility reasons, even with polyfills.)

`module`

The `module` value specifies the output module format. The *commonjs* module is still the most dominant module format, but it can be set to other values as well, e.g., `"es2015"` or `"es2020"` for ES modules, etc. NPM (Node package manager service) uses certain rules to support both `"commonjs"` and `"es2020"` modules. To use that feature, you can set the `module` value to `"nodenext"`.

```
{
  "compilerOptions": {
    "module": "es2020"
  }
}
```

`allowJs`

In case you are working on a large project with some part of codebase in Javascript, you can set this value to `true` to include JS files in the project. In such a case, Typescript will also treat *.js* and *.jsx* files as part of the program by default in addition to *.ts*, *.tsx*, and `.d.ts` files. In general, however, if you are starting a new application project, then there is really no reason to mix TS and JS source files.

```
{
  "compilerOptions": {
    "allowJs": false
  }
}
```

(When you use `"allowJs": false`, which is recommended, you do not generally add JS files into the source code repository. The JS files are generated files (e.g., the compiler output), and they need not be included in the version control system. This can be easily achieved by

adding one line **.js* in the *.gitignore* file, for instance. When you mix TS and JS source files with `"allowJs": true`, on the other hand, you will need to be a bit more creative in terms of where to put the source JS files vs the generated JS files, etc.)

`alwaysStrict`

Most Javascript developers use the `"use strict"` mode. Many modern ECMAScript features use the `"use strict"` mode by default, for example, in class definitions, or in ES modules, with no way to "turn it off". Typescript is based on this strict mode variant of Javascript. Typescript, by default, also emits JS code in the `"use strict"` mode, with `"alwaysStrict": true`.

```
{
  "compilerOptions": {
    "alwaysStrict": true
  }
}
```

Type checking

Not surprisingly, there are many options that affect the behavior of *tsc* in terms of static type checking. In theory, you can tweak each of these options to get the "perfect variant of Typescript" that you want to use. In practice, however, that is an option that you should rarely use (unless you are learning Typescript and need a "training wheel").

All these settings are primarily useful for transitional purposes. For example, for migrating an existing JS project to Typescript, or for integrating legacy code into a TS project, etc. (BTW, from the Typescript developer's perspective, all Javascript code is "legacy". ☺) We will look at a few of the important options next.

2.5.3. `strict`

Setting `strict` to `true`, or using *--strict* flag with *tsc*, enables all strict type-checking options, e.g., `strictNullChecks`, `noImplicitAny`, `noImplicitThis`, `strictBindCallApply`, `strictFunctionTypes`, `strictPropertyInitialization`, `useUnknownInCatchVariable`, and possibly any future additions related to the strict type checking. Each individual option can be selectively disabled.

```
{
  "compilerOptions": {
    "strict": true
  }
}
```

As stated, except as a learning aid or migration tool, all `strict` options should be enabled as a general rule, e.g., for all future Typescript projects, through this one setting, `"strict": true`. We assume that `strict` is set to `true` throughout this book, and we do not discuss other possible settings, and their implications, in this book.

2.5.4. `strictNullChecks`

This setting affects whether `null` or `undefined` can be a valid value of other types. With `"strictNullChecks": true`, a variable of the `string` type, for example, cannot be assigned `null` or `undefined`, which can prevent us from running into some (rather common) null-related errors.

```
{
  "compilerOptions": {
    "strictNullChecks": true
  }
}
```

2.5.5. noImplicitAny

The broadest possible type in Typescript is `any`, as we discuss later. When you completely lack type information on a variable, for example, you can annotate it with `any`, which really means that its type can be anything. With `"noImplicitAny": true`, even then, you will still have to *explicitly* annotate it with `any`. That is, you cannot expect the compiler to automatically infer a type to be `any`.

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

2.5.6. noImplicitThis

With `"noImplicitThis": true`, which is also a part of `"strict": true`, Javascript's `this` needs to be explicitly type-annotated. This is explained later with respect to the `this` function parameter.

```
{
  "compilerOptions": {
    "noImplicitThis": true
  }
}
```

Chapter 3. Module System

3.1. ES Modules

Historically, many different formats of modules have been used in Javascript. For example, CommonJS is one of the most widely used module formats, which uses the global variable `exports` for exporting, and the `require` function for importing. CommonJS has been used as the default module format in the NPM package repository.

Since ES2015, however, the Javascript community has been (slowly) moving more toward the ES module system, which uses the Javascript keywords `export` and `import` for exporting and importing, respectively. As of 2023, when this book is written, most Javascript runtimes, including all major Web browsers, now support ES modules. You can also publish your NPM package in the ES module format (e.g., by setting `module` to `es2020` or `nodenext`).

In ECMAScript, a file containing a top-level `import` or `export` is a module. Everything else is just a "script". In a module, `"use strict"` is implicitly declared. Typescript follows the same convention. Unless a Typescript file contains an `import` or `export`, it is considered a script, and the script is executed in the global scope. As indicated, in Typescript, `"use strict"` is always implied.

3.2. Typescript Modules

Syntactically, Typescript's module system more or less follows the ES module system. But, as indicated earlier, the actual module output format is controlled by the *module* configuration value in the *tsconfig.json* file. In particular, Typescript supports a number of different module formats, including a few different variants of ES modules, `es6`/`es2015`, `es2020`, `es2022`, and `esnext`. In addition, it

supports `commonjs`, `amd`, `umd`, `system`, `node16`, and `nodenext`. How exactly these values are interpreted by Typescript is determined by the module resolution rules, which we do not include in this book.

If you are familiar with the ES module system in Javascript, you can skip the rest of this chapter.

3.3. Module Exports

The `export` declaration is used to export a reference to an object or value from a module so it can be used by other scripts or modules. There are two kinds of exports, "regular" exports and `default` exports:

```
export const catLives: number = 9;        ①
```

① A type can be inferred by Typescript in this example, but we will sometimes include explicit type annotations in this book for illustration.

You can export any number of objects, functions, or primitive values using this syntax. On the other hand, a module can have at most one default export.

```
export default <number>9;                 ①
```

① An explicit type assertion is not required. That is, `export default 9;` should have sufficed in this example.

Note that the default export does not require a name.

3.3.1. (Regular) exports

The non-default exports can take a few different forms:

Variable `export`

```
export var mysteryText: string;
export var badNumber: number = 13, goodNumber: number = 7;
```

You can export one or more variables in one `export` statement.

Declaration `export`

You can export function, class, constant, and `let` declarations as follows:

```
export const INTERVAL_SECS: number = 100;
export let tempDir: string = '/tmp', tempFile: string;
export function catchABreak() { }
export class BreakCatcher { }
```

You can export essentially multiple names using these two export syntaxes, e.g., by using an object with properties. For instance,

```
export let obj1: { key: string }          ①
    = { key: 'value' };                   ②
```

① The type of the variable `obj1` is `{ key: string }`.

② `obj1` is initialized with an object `{ key: 'value' }`.

Or, using type inference,

```
export let obj1 = { key: 'value' };
```

You can also use destructuring to rename select properties of an object. For instance, without explicit type annotation,

```
const obj2 = { key1: 'v1', key2: 'v2' }  ①
export var { key1, key2: altKey } = obj2;
```

① The type of `obj2` is `{key1: string, key2: string}`. The object literal type is explained later in the Object Types chapter.

Name list `export`

```
class Children {};
var child1: string, child2: number = 42, child3: boolean;
export { Children };
export { child1, child2, child3, };      ①
```

① Note that the exported object has a type `{ child1: string, child2: number, child3: boolean }`.

3.3.2. Default exports

The `default export` declaration gives a certain syntactic convenience to the importing modules. The `default` export can take a few different forms:

Default function `export`

You can export a function as a module's default export in one of the two ways:

```
export default function() { };
```

This statement exports an anonymous function as the module's default export. Likewise, a named function can be exported as the module's default:

```
export default function catchMeIfYouCan() { };
```

Functions can also be default exported as follows:

```
function catchABreak() { };
export default <() => void>catchABreak;  ①
```

① The type of the `catchABreak` function is `() => void`. The function types are discussed later. The type assertion is unnecessary in this example.

Default class `export`

You can also export a `class` as a module's default export as follows:

```
export default class BreakCatcher { };
```

In the same way, we can also export a `class` defined earlier as the module's `default`:

```
class BreakCatcher { };
export default BreakCatcher;              ①
```

① The type of the class `BreakCatcher` is `BreakCatcher`. That is, a class declaration creates a new type.

3.4. Module Imports

The (static) `import` declaration is used to import the names exported by other modules. There is also dynamic `import()`, which was introduced in ES2020. We will not discuss dynamic imports in this book. There are two kinds of `import` statements.

3.4.1. Module `import`

You can just import a module:

```
import './my-other-module';                ①
```

① This statement imports a module in a file *my-other-module.ts* in the current directory.

This form of `import` declaration is for "side effects" only. The statements in the imported module will be executed in the context of the importing module.

3.4.2. `import - from` Declaration

We import specific names and definitions exported from other modules using the `import - from` statement. This is the more typical uses of the `import` declarations.

```
import X from './his-module';
```

There are a few different kinds of `import - from` declarations. The ways in which we can import the names depend on how they are exported from the imported module.

3.4.3. Default `import`

```
import catchMe from './function-module';
```

In this case, the `'function-module'` module has a default export, and we are naming the default exported object as `catchMe`. The name is arbitrary, and any valid Typescript identifier will do.

3.4.4. Name list `import`

```
import { child1, child2, } from './her-module';
```

If the `'her-module'` module exports `child1` and `child2`, among others, then we can import them by their names. We can also use the JS `as` keyword to rename any of them to use different names, or aliases.

```
import { child1 as First, child2 as Second } from './her-
module';
```

3.4.5. Namespace `import`

You can import all exported items from a given module using the wildcard `*` syntax.

```
import * as AlienModule from './their-module';
```

For example, if the imported module, from the file `'./their-module.ts'`, is exporting a function named `phoneHome`, then we can now refer to it as `AlienModule.phoneHome`. Note that the `as` clause with the namespace alias is required in this syntax.

3.5. Typescript Namespaces

Typescript's construct called the `namespace` is another module format, which predates the ES module standard. Now that the ES module system is becoming more widely used, the use of TS-specific namespaces is discouraged. We do not discuss Typescript namespaces in this book.

Chapter 4. Variables

Variables can be declared using keywords `const`, `let`, or `var`. In the non-`"use strict"` mode, variables do not need to be first declared before they can be used. But, as indicated, we do not use Javascript's non-strict mode in this book.

4.1. The `const` Declaration

The `const` declaration in Javascript declares a variable and assigns an initial value. The variable cannot refer to anything else other than this initial value. For example,

```
> const animal = "giraffe"
> animal
'giraffe'
```

Attempting to assign a new value to a `const` variable will cause an exception.

```
> animal = "elephant"
Uncaught TypeError: Assignment to constant variable.
```

Unless there is a specific reason otherwise, the `const` declaration should always be the first choice among the three different kinds of declaration syntax.

In Typescript, a `const` variable can be declared in the same way:

```
const plant: string = "oak";              ①
// plant = "pine";                        ②
```

① We do not need to specify a type in cases like this, as we will further discuss later in the Type Annotation chapter.

② The static type analyzer in the IDE would not even let us write an incorrectly typed code. Note that, by convention, we will comment out code that will raise a compile-time error.

Note that the terms like const or constant (or, readonly or immutable, and so on) can have different meanings across different programming languages, and/or in different contexts. In Javascript and Typescript, it simply means that variables declared with `const` cannot be used to refer to values other than the ones they are initially assigned to.

4.2. The `let` Declaration

The `let` declaration in Javascript is essentially the same as the `const` declaration, except that

- The `let` variable need not be explicitly initialized in the declaration, and
- The `let` variable can be used to reference different values throughout its lifetime.

When a variable is not explicitly initialized in the `let` declaration, its initial value is `undefined`.

```
> let insect
> insect
undefined
```

In Typescript, every variable needs to be associated with a type, and its type cannot change (even though its associated value can).

```
let vegetable;                          ①
```

```
let fruit: string;                    ②
// console.log(fruit);                ③
let flower: string = "rose";          ④
// flower = 3;                        ⑤
```

① A `let` variable without an initial value will be assigned `undefined`, and its type will be inferred to be the broadest possible type in Typescript, `any`.

② We can declare a `let` variable with an annotation with a type.

③ But, this variable cannot be used before assignment. This is because the Javascript's default initial value `undefined` is not compatible with the specified type, `string` in this example. Note that we always assume that all Typescript strictness settings are enabled in this book, including `strictNullChecks`.

④ In case of a `let` variable declaration with an initial value, the variable may not have to be annotated if the inferred type is suitable. In this particular example, the type annotation, `string`, is redundant.

⑤ In Javascript, this might have been allowed. But, in Typescript, assigning a value to a variable with an incompatible type, e.g., `string` vs `number`, will throw a compile-time error.

4.3. The `var` Declaration

The `var` declaration is similar to the `let` declaration, but they have different scoping rules. Generally speaking, the (newer) `let` declaration syntax is preferred over the (older) `var` declaration syntax. For top-level variables, however, there is little difference, in practice, between `var` and `let`.

The readers are encouraged to consult a Javascript reference for the Javascript variable scoping rules, and in fact, for any Javascript topic that is not covered in this book.

In all three different types of declarations, multiple variables can be declared in one statement. For example,

```
> var animal = "rabbit", count = 10;
> console.log(animal, count);
rabbit 10
```

In Javascript, we can reassign values of different types:

```
> animal = 20, count = "turtle";
> console.log(animal, count);
20 turtle
```

In case of Typescript, with a type annotation (which is unnecessary in this example),

```
var flower: string = "tulip",
    count: number = 6;
console.log(flower, count);
```

Once the type is determined for a variable, either through an explicit type annotation or type inference, it cannot change. Except for the `const` variables, the variables declared with `var` or `let` can still be modified, that is, they can refer to different values as long as their types are compatible.

```
flower = "rose";
count = 12;
console.log(flower, count);
```

Chapter 5. Basic Types

We will cover some simple and fundamental types in this chapter. They are not only the most commonly used, but they also form the building blocks of more advanced and more complex types in Typescript.

5.1. Javascript Types

Javascript includes eight fundamental types: `boolean`, `number`, `bigint`, `symbol`, `string`, `undefined`, `null`, and `object`. The first seven types are primitive types, and everything else is `object` in Javascript. Arrays are objects, and functions are objects.

Javascript is an object-based programming language, and objects play arguably more fundamental roles than types (or, "classes"). Arrays and Functions are builtin objects. In fact, there are predefined objects corresponding to all primitive types, except for `null` and `undefined`. That is, there are `Boolean` for `boolean`, `Number` for `number`, `BigInt` for `bigint`, `Symbol` for `symbol`, and `String` for `string`. Javascript will automatically convert values of the primitive types to the corresponding objects at run time, if needed. This is sometimes called auto-boxing in some other programming languages.

Furthermore, Javascript defines a number of additional builtin, or global, objects. (We often use the names Javascript and ECMAScript interchangeably in this book even when the use of one over the other may be preferred.) For example, Javascript includes builtin functions such as `parseInt` and `parseFloat`, core language objects such as `Error` and `Promise`, collection objects such as `Map` and `Set`, and other builtin objects such `Date`, `Math`, `RegEx`, and `JSON`.

Typescript defines a number of types corresponding to some of these builtin objects in Javascript. As a matter of fact, any object which is a constructor function in Javascript can be viewed as a type since it can

create multiple structurally equivalent objects, or "instances", based on its prototype. The modern Javascript now uses `class`, and, in Typescript, every `class` defines a new type.

5.1.1. The `typeof` operator

Javascript has a `typeof` operator, which takes an expression and returns its type at run time, *as a string*, that is, one of `'boolean'`, `'number'`, `'bigint'`, `'symbol'`, `'string'`, `'undefined'`, `'function'`, or `'object'`.

For example,

```
> typeof true
'boolean'
> typeof 2023
'number'
> typeof 3.1415
'number'
> typeof "Wizard College"
'string'
> typeof (() => undefined)
'function'
> typeof null
'object'
```

Note that, although `null` is considered a separate type in Javascript, `typeof(null)` simply returns `'object'`.

5.1.2. The `instanceof` operator

The binary `instanceof` operator takes an object (LHS) and a constructor (RHS) and it returns `true` if the prototype property of the constructor appears anywhere in the *prototype chain* of the given object. It returns `false` otherwise.

```
> function Bot(model) {                    ①
...     this.model = model
... }
> Bot instanceof Function                  ②
true
> Bot instanceof Object                    ③
true
> const chatbot = new Bot('Chatbot');      ④
> chatbot instanceof Bot                   ⑤
true
> chatbot instanceof Object                ⑥
true
> chatbot instanceof Function              ⑦
false
> "Hello World Cup!" instanceof Object     ⑧
false
```

① A simple Javascript constructor function.

② `Bot` is a `Function` object.

③ Every object in Javascript is an `Object`.

④ One can create an instance of `Bot` using the `new` operator.

⑤ The `chatbot` object is an "instance of" `Bot`.

⑥ The `Object` constructor appears in every object's prototype chain.

⑦ `chatbot` is not a `Function`.

⑧ A value of a primitive type is not an instance of `Object`.

5.2. Strict Equality

Javascript has two equality and two inequality operators: Equality `==` and strict equality `===`, and inequality `!=` and strict inequality `!==`. The strict versions take the operands' (run-time) types into account whereas the non-strict versions do not.

5.2. Strict Equality

In Typescript, since all values and variables are associated with (static) types, the non-strict versions of equality `==` and inequality `!=` operators work *differently* than in Javascript.

For example,

```
> 1 === "1"                                    ①
false
> 1 == "1"                                     ②
true
```

① In Javascript, the use of strict equality and inequality operators is generally recommended.

② This returns `true`.

In Typescript, however, the expression `1 == "1"` is invalid because the operands have two different static types, i.e., `number` vs `string`. Hence, in general, using non-strict equality and inequality operators is often safe in Typescript.

Note, however, that, to be perfectly safe, the use of strict equality and inequality operators may still be preferred. For instance, the following example shows one gotcha of using non-strict equality and inequality operators, even in Typescript.

```
let one1: number = 1;
let one2: unknown = "1";                       ①
console.log(one1 == one2);                     ②
```

① The `unknown` type bypasses strict type checking.

② This will print `true`, which is most likely not the result that the Typescript developer expected.

5.3. Primitive Types

5.3.1. The `boolean`, `number`, and `string` types

Typescript supports all primitive types in Javascript, including `boolean`, `number`, and `string`.

- The `boolean` type has two values, `true` and `false`.
- The `number` type includes all 64 bit integer values and 64 bit floating point values.
- The `string` type represents strings such as "Hello Wizard!".

```
const thumbsUp: boolean = true;
const year: number = 2023;
const totalHours: number = 10040.25;
const planet: string = "Mars";
```

5.3.2. The `bigint` Type

In addition to `number`, Javascript also includes the `BigInt` type (since ES2020), which represents the "infinite precision" integer numbers. For example,

```
> typeof BigInt(1_000_000)
'bigint'
> typeof 1_000_000_000n                    ①
'bigint'
```

① The `BigInt` literal syntax can be used only if the *target* is set to `es2020` or later. Otherwise, you can use the constructor syntax, e.g., `BigInt(1_000_000_000)`.

The type of `BigInt` values in Typescript is `bigint`.

```
const oneMillion: bigint = BigInt(1_000_000);
const oneBillion: bigint = 1_000_000_000n;
```

5.3.3. The `symbol` type

Since ES2015, Javascript includes the `Symbol` type, which can be used to create globally unique references. One can create new and unique symbols as follows:

```
> const myMagic = Symbol("huff")
> const yourMagic = Symbol("huff")
> typeof myMagic
'symbol'
> typeof yourMagic
'symbol'
```

The type of `Symbol` values in Typescript is `symbol`.

```
const myMagic: symbol = Symbol("huff");
const yourMagic: symbol = Symbol("huff");

if (myMagic !== yourMagic) {
    console.log("We are not the same.");
}
```

5.4. Literal Types

Literal values of `boolean`, `number`, or `string` can be used as types, often in union types. For instance,

```
const hello = "Hullo~~~";
```

In this example, the variable `hello` can represent only one specific string `"Hullo~~~"` throughout the program, and hence its literal value represents its type in Typescript. The above statement can be written as follows using an explicit type annotation:

```
const hello: "Hullo~~~" = "Hullo~~~";    ①
```

① In this case, the literal type `"Hullo~~~"` is sort of a subtype of `string`. Note that literal types and their values have the same representations. Whether a literal is used as a type or a value depends on the *context*.

5.5. The any Type

The `any` type is the broadest type in Typescript (as in "any type"), and it is sort of a supertype of all types, builtin or user-defined, in Typescript. But, `any` is a special type and its use is generally discouraged except during development. For example, when you need a type and you don't know what type to use, you can temporarily use this type, `any`, e.g., to avoid type checking errors, etc.

The `any` type implicitly includes all possible properties and all possible methods. A value of type `any` can be used anywhere, it can be called like a function, or it can be assigned to, or from, any value of any type. That is, Typescript effectively disables all static type checking as long as `any` values are involved (e.g., as if it is a Javascript code). For example,

```
let dubious: any = { speed: 65 };        ①
let rank: number = dubious;              ②
console.log(rank, typeof rank);
dubious = 2100;                          ③
console.log(dubious, typeof dubious);
dubious.drive(dubious.velocity);         ④
```

① We explicitly declare the variable `dubious` as the `any` type for illustration.

② Although `rank` is declared as `number`, this is not enforced at compile time when an `any` value is involved. After the assignment, at run time, the type of `rank` is `object`.

③ We can assign values of any type to a variable of `any` type. At run time, after this assignment, the type of `dubious` will be `number`.

④ Although `dubious` does not have a property `velocity` or a method `drive`, it will still compile. It will, however, throw a run time error, just like a plain Javascript code.

As this simple example illustrates, the use of `any` negates virtually all the benefits of using Typescript. Note that since we (always) use `"noImplicitAny": true`, `any` is never automatically inferred by Typescript. When needed, `any` should be explicitly specified, e.g., as a temporary measure.

In terms of type compatibility:

- Any values of any type can be assigned to a variable annotated as `any`, and
- The values of type `any` can be assigned to variables of any type, except for `never`.

5.6. The `unknown` Type

The `unknown` type is a (type-safe) doppelganger of `any`. `unknown` is the same as `any` in that it also represents the broadest type in Typescript, but unlike `any`, variables of the `unknown` type are statically type checked.

- Any values of any type can be assigned to a variable annotated as `unknown` since it is the broadest type,

- Values of `unknown` can be assigned to a variable of type `any` or `unknown`. But, it cannot be assigned to variables of any other type without type assertion or narrowing, and
- No properties or methods of an `unknown` value can be accessed without first asserting or narrowing to a more specific type.

For example,

```
let dark: unknown = { shade: "gray" };   ①
dark = "chocolate";                      ②
console.log(dark, typeof dark);
```

① We are declaring `dark` as the `unknown` type so that it can be used to refer to values of any type.

② We can assign value of any type to the variable `dark`.

```
let dark: unknown = { shade: "gray" };   ①
// let cost: number = dark;              ②
let cost: number = dark as number;       ③
console.log(cost, typeof cost);          ④
```

① The same example as above.

② This will fail static type checking since a value of the "broader" type cannot be assigned to a variable of the "narrower" type.

③ We can "trick" the static type checker by using type assertion, for instance.

④ At run time, the type of `dark`, and hence that of `cost`, is still `object` (not `number`).

Another example,

```
let dark: unknown = { shade: "gray" };
```

```
// console.log(dark.shade);                          ①
console.log((dark as { shade: string }).shade);      ②
console.log((dark as { pray: () => void }).pray());  ③
```

① Although `dark` has the property `shade`, it cannot be directly accessed since it is of the `unknown` type.

② We can use type assertion to access `shade`.

③ We can also "trick" Typescript to think there is something that does not exist. This statement will pass type checking, but it will fail *at run time* since `dark` does not have a method named `pray`.

5.7. The `null` and `undefined` Types

Javascript has two primitive values used to indicate an absent value or an uninitialized value, `null` and `undefined`. TypeScript has the corresponding types by the same names. The `null` type has a single value `null`, and the `undefined` type likewise has a single value `undefined`. These types are, therefore, called the "singleton types".

Note that `null` and `undefined` are two separate and distinct types in Typescript.

```
let alwaysNull: null = null;              ①
alwaysNull = null;                        ②
// alwaysNull = undefined;                ③
let alwaysUndefined: undefined;           ④
alwaysUndefined = undefined;              ⑤
// alwaysUndefined = null;                ⑥
```

① We declare `alwaysNull` as a `null` variable with an initial value `null`.

② This variable `alwaysNull` can be assigned (the same) `null`, but nothing else.

③ We cannot even assign `undefined` to this variable.

④ We declare another variable `alwaysUndefined` as the `undefined` type. Typescript initializes any variable with `undefined` which has no explicit initial value, and hence the initial value is consistent with the type annotation.

⑤ This variable is no good other than referring to one and only one `undefined`.

⑥ It is illegal to assign any other value, including `null`, to this variable.

In Javascript, although `null` and `undefined` are two distinct values (and types), they are largely interchangeable (as long as they are used consistently). The biggest (conceptual) difference is that `null` is explicit whereas `undefined` is more implicit. That is, when you declare a variable *without* an initial value, its initial value is `undefined`. If you use a `return` statement *without* an explicit return value in a function, that is equivalent to returning `undefined`.

```
> (function a() { return })()            ①
undefined
> (function b() { return undefined }) ()
undefined
```

① In Javascript (and, hence in Typescript), there is absolutely no semantic difference between `return;` and `return undefined;`.

If you want to treat `null` and `undefined` as more or less the same, then you can use a union type, e.g., `null | undefined`.

Note that we always assume that `"strictNullChecks": true` in this book. Setting `strictNullChecks` to `false` will result in different behavior of Typescript type checking when it comes to `null` or `undefined`.

5.8. The never Type

The never type has no valid value, and it indicates values that should not occur. never is the narrowest type. That is, it can be viewed as a subtype of all types defined in Typescript.

never can be used as the return type of a function that never returns. For example,

```
function oblivion(): never {
    throw new Error("Never returns");
}
```

Or,

```
const f: (() => never) = () => {          ①
    while (true) { }
}
```

① The function types, e.g., `() => never` in this example, are discussed later.

In terms of assignability,

- No values of any type, except never itself, can be assigned to a variable annotated as never, and
- Values of never can be assigned to variables of any type.

5.9. The void Type

The void type is typically used as a function return type to indicate that the function does not return any value. For example,

```
function empty(): void {
    return;
}
```

In this context, `void` is more like `undefined`. In fact, `void` is generally considered a slightly broader type than `undefined`.

In terms of assignability,

- Values of `void`, `undefined`, and `never` types, but of no others, can be assigned to a variable annotated with `void`, and
- Values of type `void` can be assigned to variables of `void`, `any`, or `unknown` types, but to no others.

5.10. The `object` Type

The special type `object` is a supertype of any type that represents *objects* in Typescript (that is, excluding values of primitive types).

For example,

```
const obj1: object = { prop: "value" };  ①
console.log(obj1 instanceof Object);      ②
const obj2: object = [1, 2, 3];           ③
console.log(obj2 instanceof Array);
const obj3: object = new String("huh?"); ④
console.log(obj3 instanceof String);
```

① The `object` type is "compatible" with any object. It is sort of like an ultimate base type of other object types.

② Note that `Object` is a builtin global object in Javascript (which is at the top of the prototype chain for other objects). This statement will print `true`.

③ An array is also an object, and hence it is (indirectly) of the `object` type. Both `obj2 instanceof Array` and `obj2 instanceof Object` will return `true`.

④ A `String` object (but, not a `string` primitive value) is also an object, and hence it is (indirectly) of the `object` type. Both `obj3 instanceof String` and `obj3 instanceof Object` will return `true`.

Denoting specific object types in Typescript is discussed later in the Object Types and Interfaces chapters. In addition, the Typescript Class, which is based on the ES class, provides a convenient way to declare a new type and to easily create one or more objects of the same type.

5.11. Function Types

Functions are one of the most basic building blocks in *any* programming languages, including Javascript (and, hence Typescript). In Javascript, every function is a `Function` object, whose prototype chain includes both `Function.prototype` and `Object.prototype`.

For example,

```
> function shout() {
...     console.log("HELLO, WORLD!")
... }
> shout instanceof Function
true
> shout instanceof Object
true
```

Typescript does not define a separate overall function type that corresponds to Javascript's `Function` object. We discuss how to annotate a function with a specific function type later in the book, e.g., in the function annotations section, and the Function Types chapter.

For instance,

```
function yell(level: number): string {    ①
    switch (level) {                      ②
        case 1: return "hello world";
        case 2: return "Hello World";
        default: return "HELLO WORLD";
    }
}
```

① The function `yell` takes a `number` argument and returns a `string` value.

② Any valid Javascript statement is also a valid Typescript statement.

5.12. Array Types

Arrays are used to store a sequence of items as a collection, e.g., as a single object. All arrays in Javascript are `Array` objects, and they support common array operations such as subscripting, etc., as in many other programming languages.

For example,

```
> const a = [1, 2, 3]                     ①
> typeof a                                ②
'object'
> a instanceof Array                      ③
true
> a instanceof Object
true
> Array instanceof Object
true
```

① An array literal syntax. An array in Javascript can also be created using the (overloaded) `Array` constructor function.

② Javascript arrays are just objects.

③ In Javascript, the `instanceof` operator, which checks the constructor prototype chain, can be usually more useful than the `typeof` operator.

In Typescript, there is a different array type for each different element type. For instance,

```
const abc: string[] = ['a', 'b', 'c'];   ①
```

① The type of `abc` is `string[]`, an "array of string elements". Array in Typescript is a generic type. In fact, `string[]` is a shorthand for `Array<string>`.

Unlike in Javascript, arrays in Typescript must be "homogeneous" for the purposes of type annotations. That is, all of their elements should belong to a single type, possibly including the broadest type `unknown` (or, `any`).

```
const xyz: unknown[] = ['x', 7, false];  ①
```

① The array `xyz` includes three elements of `string`, `number`, and `boolean`. We annotate it as `unknown[]` in this example. A "better" type would have been `(string | number | boolean)[]`, using the union type of all its element types.

Typescript's array types are further discussed in the Arrays chapter.

5.13. Tuple Types

Typescript includes additional array-like collection types called the tuples. In fact, they are just arrays in Javascript. For example,

```
> const t = ["argentina", 2022, "quatar"]
> t instanceof Array
true
```

The array `t` is no different than `a` in the Javascript example of the previous section. In Typescript, however, objects like `t` can be best typed as a tuple (e.g., rather than an array type with broad element types).

```
let winner: [string, number]                ①
    = ["france", 2018];
winner = ["argentina", 2022];               ②
```

① We declare the variable `winner` as a two-element tuple type, `[string, number]`.

② We assign another tuple, with the same type, to `winner`.

An array in Typescript can grow or shrink. That is, its size can change. Tuples in Typescript are fixed size, with fixed element types, at least, conceptually. In practice, however, since both arrays and tuples become Javascript arrays upon transpilation, Typescript's tuple support is not perfect. For example,

```
winner.push(2026);                          ①
console.log(winner);
```

① Typescript does not prevent us from adding more elements to a (fixed-size) tuple. At run time, `winner` will end up being `['argentina', 2022, 2026]`, whose proper type would have been `[string, number, number]`.

Typescript tuple types are further discussed later in the book, including readonly tuples.

5.14. Enum Types

Enum, or enumeration, types are often used to define a set of related constants. Typescript's `enum` is an addition to the language (beyond Javascript). When Typescript code is transpiled to Javascript, the `enum` values are converted to proper Javascript code (e.g., using `const`).

For example,

```
enum GOAT { "pele", "maradona" }          ①
let goat: GOAT;                           ②
goat = GOAT.pele;                         ③
console.log(goat);                        ④
```

① This `enum` declaration defines a new type, `GOAT`. The valid values for this type are `pele` (index 0) and `maradona` (index 1).

② We declare a variable `goat` with the type `GOAT`.

③ We can assign a value of the `GOAT` type to this variable.

④ This will print out `0`, the numeric value of `GOAT.pele`.

Most programming languages include some kind of enum or enum-like constructs. Enum has been a part of Typescript from the very beginning. At this point, however, there are many different ways to achieve the same things, and the use of `enum` in Typescript is not particularly encouraged.

Note that, at some point in the future, Javascript may introduce its own enum construct, which may or may not be compatible with Typescript's enum.

Chapter 6. Type Aliases

Type aliases are primarily used to provide names to (anonymous) type literals. But they can also be used to assign new/different names to existing named types.

Type alias declarations, using the Typescript keyword `type`, are syntactically similar to variable declarations (e.g., using `const`, `let`, or `var`), and type aliases have similar semantics to variables, e.g., in terms of scoping and shadowing, etc. But, type aliases are a purely compile-time construct. As an example, one can declare type aliases for primitive types as follows:

```
type Tax = number;                    ①
const iou2023: Tax = 100.0;           ②
```

① The type `Tax` is just an alias to the primitive type `number`.

② The type of the variable `iou2023` is `number`. Type aliases can be useful for code documentation purposes. That is, in this example, the number `100.0` has something to do with `Tax`.

Here's a slightly more complicated example:

```
type Code = 400 | 401 | 404;          ①
let clientCode: Code = 404;           ②
```

① `Code` is a type alias to a union type with three literal types, `400`, `401`, and `404`.

② `404` is a valid value for the union type `Code`, and hence it can be assigned to a variable of type `Code`.

One of the most common uses of type aliases is to give a name to an object literal type. For instance,

```
type Point2D = {                          ①
    x: number;
    y: number;
};
let point: Point2D;                       ②
point = { x: 1.0, y: 2.0 };
```

① All type alias declarations have the same syntax. The keyword `type` followed by a name, an equal sign `=`, and a target type, which is an object literal type, `{ x: number; y: number; }`, in this example.

② Type aliases can be used just like (regular) types.

Note that object literal types with type aliases and interface types have overlapping use cases. We discuss this further throughout the book.

Here's a somewhat convoluted example:

```
type ID = number;                         ①
const myID: ID = 42;                      ②
{                                         ③
    type ID = string;                     ④
    const yourID: ID = "forty two";       ⑤
    console.log(myID, yourID);
}
```

① We create a type alias `ID` referring to the `number` type.

② The type of `myID` is `number`.

③ A pair of angular brackets `{}` creates a new block, and a new scope, in Javascript (and, hence in Typescript).

④ In the inner block, we declare a type alias with the same name `ID`. Here, `ID` is an alias to `string`. The alias `ID` declared in the outer scope is "shadowed" at this point.

⑤ The type of `yourID` is `string`.

Chapter 7. Type Annotations

7.1. Variable Annotations

When a variable is declared using `var`, `let`, or `const`, a type can be explicitly specified for the variable with the *variable* `:` *type* syntax. For example,

```
var apple: string;                    ①
let orange: number = 100;             ②
const pear: boolean = true;           ③
```

① As far as Typescript is concerned, this variable `apple` cannot be used to refer to any value which is not of the `string` type.

② The variable `orange` can refer to different numbers, but only numbers.

③ The value of the variable `pear` is always `true` throughout the execution of the program, and nothing else.

In the second variable declaration, the type annotation is not entirely needed because the Typescript compiler can *infer* the type of `orange` based on its initial value, `100`, which is `number`. Hence we could have done

```
let orange = 100;                     ①
```

① The type of `orange` is inferred to be `number`.

More or less the same logic applies to the declaration for `pear`, which is initialized with `true` and hence the type can be inferred. In case of `const` variables, however, the type is inferred to be the literal type of the initial value, if feasible. For example, in the following,

```
const pear = true;
```

The type of `pear` will be inferred to be a literal type `true`, not `boolean`. But, since `true` is assignable to `boolean`, in practice, these two declarations, with and without explicit type annotations, would be more or less equivalent to each other.

In general, when a variable is initialized with a specific value of a specific type, and that type is the desired type, type annotation is superfluous. If the compiler-inferred type turns out to be not the desired type, then the explicit type annotation is still needed. For example,

```
let pineapple: (string | number) = "Sweet and sour";      ①
pineapple = 666;                                          ②
```

① The intended type of `pineapple` cannot be inferred from the initial value alone.

② We end up assigning a number to this variable.

7.2. Function Annotations

A function type annotation can be viewed as an extension of variable type annotation. For functions, their parameters and their return values need to be annotated.

For example,

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

This function `add` takes two arguments of the type `number` and it returns a value of `number`. Javascript functions can also be declared anonymously:

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

Or, alternatively, using the fat arrow function syntax:

```
const add = (a: number, b: number): number => a + b;
```

The parentheses `()` around the function parameters, in Typescript, are required even with one parameter when the parameters are type-annotated. Note that the type of the variable `add` in this example is `(a: number, b: number) => number`. Function types are discussed later.

When a function is called, the Typescript compiler checks the types of the arguments and that of the return value so that they are compatible with the annotated types. In general, type annotations on functions are more important, especially type annotations on parameters. They often carry information not only to the Typescript compiler but also to other developers, or API clients, who use the function.

Unlike in the case of variable type annotations, functions are usually declared and called in two different places, and therefore their relevant types are not easily inferrable at the point of function declarations. An exception is an (anonymous) function that is defined and called at a single location. For instance,

```
const sum = ((a, b) => a + b)(3, 4);
console.log(sum);
```

In this example, the arrow function, `(a, b) => a + b`, is declared and called in one statement. Based on the provided arguments, `3` and `4` in this case, the types of `a` and `b` can be inferred to be both `number`. The type of the return value can also be `number` since addition of two numbers yields a number.

Another related scenario is when an (anonymous) function is declared and passed in as an argument to a higher-order function. If the parameter type of the higher-order function is known, the arrow function argument, for instance, does not need to be explicitly annotated.

For example,

```
const m1 = [1, 2, 3].map(a => 2 * a);
const m2 = [1, 2, 3].map(
    function (a) {
        return 2 * a;
    }
);
```

In this example, the type of `a` in both cases can be easily inferred to be `number` since the `map` function acting on a `number` array expects a function of type `(a: number) => number` as an argument.

These two functions could have been annotated as `(a: number): number => 2 * a` and `function (a: number): number { return 2 * a; }`, respectively.

Chapter 8. Assertions

8.1. Type Assertions

In certain situations, you, as a developer, may know that the actual type of a given variable may be more specific than the one statically annotated (or, inferred). In such a case, you can explicitly declare the type of the variable as this more specific type so that the compiler can utilize that information. This is called the type assertion in Typescript.

Despite the name, it is not a runtime assertion. Like type annotations, type assertions are removed at compile time, and it won't affect the runtime behavior of the generated Javascript code.

There are two syntactic forms of type assertion. First, we can use Typescript's `as` operator.

For example,

```
let puzzle: unknown;                        ①
let input = "forty two";
puzzle = input;                             ②
const upper
    = (puzzle as string).toUpperCase();     ③
```

① Let's assume that the type of `puzzle` is initially `unknown`.

② At this point, we know that `puzzle` refers to a string value.

③ By using a type assertion `puzzle as string`, we can call a string method on `puzzle`. Note that, without this type assertion, Typescript would not let us call this method on the `unknown` value.

Alternatively, we can also use the angular bracket `<>` syntax. Using the same example as above,

```
const upper
    = (<string>puzzle).toUpperCase();    ①
```

① The type assertion `<string>puzzle` is the same as `puzzle as string`. This angular bracket syntax cannot be used, however, in React *.tsx* files.

8.2. Const Assertions

A `const` assertion can be used on literal type expressions, including the object literal types, with the effect that

- No literal types in that expression should be broadened,
- Object literals get `readonly` properties, and
- Array literals become `readonly` tuples.

The syntax of `const` assertions is similar to that of type assertions, but instead of using the type name, it uses the keyword `const`.

For example,

```
let u1 = 42;                          ①
let u2 = 42 as const;                 ②
let u3: 42 = 42;                      ③
```

① The type of `u1` will be inferred to be `number`.

② With the `const` assertion, the type of `u2` is the literal type `42`.

③ This declaration with an explicit type annotation is equivalent to the declaration in the second line with the `const` assertion.

When the `const` assertion is used with an array literal value, the type becomes a `readonly` tuple:

```
let v1 = ['a', 'b'];                      ①
let v2 = ['a', 'b'] as const;             ②
let v3: readonly ['a','b'] = ['a', 'b']; ③
```

① The type of `v1` will be inferred to be `string[]`.

② With the `const` assertion, the type of `v2` is a `readonly` tuple type `readonly ['a', 'b']`.

③ This declaration with an explicit type annotation is equivalent to the declaration in the second line with the `const` assertion.

Another example with an object literal value,

```
let w1 = { age: 42 };                     ①
let w2 = { age: 42 } as const;            ②
let w3: {readonly age:42} = { age: 42 }; ③
```

① The type of `w1` will be inferred to be `{ age: number }`.

② With the `const` assertion, the type of `w2` is an object type with a `readonly` property, `{ readonly age: 42 }`.

③ This declaration with an explicit type annotation is equivalent to the declaration in the second line with the `const` assertion.

The declarations for `u2`, `v2`, and `w2` can also be written as follows using the angular bracket assertion syntax (outside of *.tsx* files):

```
let u2 = <const>42;
let v2 = <const>['a', 'b'];
let w2 = <const>{ age: 42 };
```

Note that, with `const` assertions in this example, no type annotations were needed. Typescript took the most specific types from the expressions of the `const` assertions.

8.3. Non-Null Assertions

Typescript also supports a special form of type assertion, in which `null` and `undefined` are removed from a given type. More specifically, adding a suffix `!` after an expression is effectively a type assertion that the value isn't `null` or `undefined`.

For example,

```
type NullableString = string | null | undefined;
function Length(str?: NullableString): number {
    return str!.length;                    ①
}
```

① This non-null assertion tells the compiler that we are absolutely sure that `str` cannot be `null` or `undefined` at run time.

Note that this only informs the static type checker of the non-nullness of a given expression. At run time, the expression can still end up being `null` or `undefined`, which can potentially cause a runtime error. Hence, non-null assertions, and type assertions in general, should be used with caution.

Chapter 9. Generics

Generics allows defining a set of related types using one or more type parameters. Those types that are declared as a set are called parameterized types. A particular, concrete type, from a given generic type, can be obtained by substituting type parameters with specific type arguments. (As with all other type support in Typescript, generics is a strictly build-time construct.)

9.1. Why Generics?

If you are coming from Javascript background, or if you have been mostly programming in weakly typed languages, generics can be a bit puzzling. *Why do we need it? What purpose does it serve?* These are clearly more important questions to answer even before we talk about other things like how we use it.

Let's (hypothetically) suppose that we need to create a data container that can contain one number. You can add a number and retrieve it later, like some sort of a treasure chest. (It can be viewed as a degenerate case of stack and queue data structures, but this example is just for illustration.) Here's a simple implementation:

```
class NumberHolder {                        ①
    data: number | undefined;               ②
    push(data: number): void {              ③
        this.data = data;
    }
    pop(): number | undefined {             ④
        const d = this.data;
        this.data = undefined;
        return d;
    }
}
```

① Typescript classes are discussed later.

② A public field of type `number | undefined`.

③ A public method. The `void` return type annotation is optional.

④ Typescript can infer the return type of this method based on its implementation, and hence the type annotation is redundant.

The actual implementation is not important for our discussion here, but the point is that we have created a "type-safe" one number container data structure. If you try to add anything other than a number to an instance of this `NumberHolder`, e.g., using the `push` method, the static type checker will stop you from doing that.

This type safety comes at a cost, however. Now let's suppose that we need a similar data structure for strings. What do we do? One option is to make the supported type of our current `NumberHolder` a bit broader, e.g., from `number | undefined` to `number | string | undefined`. Clearly, this poses some problems. We cannot create container instances that only hold numbers and other instances that only hold strings. They are mixed, reducing the type safety. What if we need a container for `Date` objects, or another container for `StudentRecord` custom type objects, etc.? `NumberHolder` will eventually become `AnythingHolder`, and we will lose all the benefits of using the strongly typed language.

Another option is creating a container type for each data type. For example, `StringHolder` for `string` type data,

```
class StringHolder {
    data: string | undefined;
    push(data: string) {
        this.data = data;
    }
    pop() {
        const d = this.data;
```

```
        this.data = undefined;
        return d;
    }
}
```

And, another for `Date`, and one more for `StudentRecord`, and so on and on. Clearly, this is not a scalable solution even if we are willing to ignore the fact that we are duplicating code.

Then comes generics to the rescue. You can write one *generic* implementation and use it for different data types. Here's our generic `Holder` example:

```
class Holder<T> {                          ①
    data: T | undefined;                   ②
    push(data: T) {
        this.data = data;
    }
    pop() {                                ③
        const d = this.data;
        this.data = undefined;
        return d;
    }
}
```

① Note the type parameter `T` (within angular brackets), which is essentially a placeholder for real types. The type parameter names are arbitrary to a large extent, e.g., as long as they are syntactically valid identifiers.

② You can use the type parameter as if it is a real type inside the given generic class.

③ Type inference works with the generic type parameters as well. The return type of the `pop` method is `T | undefined`, which is the type of `this.data`.

Now, we can use this generic `Holder` for `number` data,

```
const numberHolder1 = new Holder<number>();
numberHolder1.push(42);
console.log(typeof numberHolder1.pop());
```

Or, for `string` data,

```
const stringHolder1 = new Holder<string>();
stringHolder1.push("Secret of Life");
console.log(typeof stringHolder1.pop());
```

Attempting to push anything other than `number` to `numberHolder1`, or anything other than `string` to `stringHolder1`, will cause a compile time error.

We have illustrated one use of generics with data container types in this section. And, that is one of the most common use cases of generics. But, one can easily imagine the use of generics in many different scenarios in practice. In fact, generics is an essential part of any strongly typed programming languages, and Typescript is no exception.

9.2. Generic Functions

To declare a generic function, you add one or more type parameters, within a pair of angular brackets `<>`, before the function parameter list.

Here's the general syntax:

```
function fn<T1, T2>(args) { /* */ }      ①
function <T1, T2>(args) { /* */ }        ②
<T1, T2>(args) => /* */                  ③
```

① A generic function declaration with a name `fn`. One or more generic type parameters (`T1`, `T2`, ...) are specified within angular brackets after the function name.

② A generic anonymous function. An anonymous function declaration can also include one or more type parameters.

③ A generic arrow function expression. Note the position of the type parameters.

(Note that functions are described in more detail later in the Function Types chapter.)

Here's an example of a generic function:

```
function firstOrDefault<T>(list: T[], value: T): T { ①
  return list.length == 0 ? value : list[0];
}
```

① Note that `T[]` represents an array type whose element type is `T`. As we discuss in the next chapter, `T[]` is a shorthand notation for the generic array type, `Array<T>`.

This function can be called as follows:

```
const x = firstOrDefault<number>([10, 20, 30], 0);    ①
const y = firstOrDefault<string>([], "Nothing");      ②
```

① Note that the parameter `T` has been replaced with a concrete type argument, e.g., `number` in this example. The value of `x` is `10`.

② The value of `y` is `"Nothing"`.

Typescript can generally infer the generic argument types based on the supplied arguments to the functions, and hence these function calls can be simplified as follows:

```
const x = firstOrDefault([10, 20, 30], 0);
const y = firstOrDefault([], "Nothing");
```

9.3. Generic Types

We have seen a simple generic class example earlier. Generic classes are further discussed later in the Classes chapter. In case of interfaces, there are two different ways to create a generic interface. First, an `interface` can include generic functions. For example,

```
interface IFace {
    write<T>(arg: T): void;
    print<T>(arg: string): T;
}
```

Note that the two functions in this example are independent of each other in terms of the type parameters (although we use the same placeholder name `T` in both cases). That is, we can do `write<string>` and `print<number>`, or `write<object>` and `print<undefined>`, etc.

Alternatively, and more commonly, an `interface` can be explicitly made *generic*:

```
interface IFace<T> {
    write(arg: T): void;
    print(arg: string): T;
}
```

In this example, the type parameter `T` is associated with the `interface` itself, and the `write` and `print` functions will end up using the same type argument that is supplied to `IFace<T>`.

9.4. Generic Type Constraints

When a generic type parameter is specified without constraints, any type can be used for that type parameter. Although they are rather common (e.g., container types), they are still special cases. More generally, a generic function or a generic type may work for a range of types, but not for all types. (And, there are an infinite number of possible types in Typescript.)

We can use the `extends` keyword to specify *type constraints* on the type parameters, that is, to specify what kind of types can be used with this particular generic type/function. For instance, the following `longestLength` function returns the max length from the given arguments:

```
function longestLength<T extends { length: number }> ①
    (...list: T[]): number {                          ②
    return (list.length == 0) ? 0 :
        Math.max(...list.map(e => e.length));         ③
}
```

① We add a type constraint `extends { length: number }` to the type parameter `T`. This means that this function can only be used with arguments with the types which include the `length` property of the `number` type. Note that the keyword `extends` in this context has little to do with type inheritance.

② The modern Javascript's rest parameter is discussed later. The `longestLength` function is a generic *vararg* function.

③ This statement uses the higher-order `map` function, the array spread syntax, and an arrow function (which does not require type annotation, as we have seen earlier), etc. But, most importantly, with regards to the current example, the type of `e` is `T extends { length: number }`. Without the type constraint, `e.length` would not have worked for general type arguments.

Here's an example usage of this function:

```
const maxLen = longestLength(                    ①
    [3, 5],                                      ②
    ['a', 'b', 'c'],
    { length: 10 },                              ③
);
console.log(maxLen);                             ④
```

① You can only use arguments which have the `length: number` property with this function. For any other values that do not satisfy the type constraint, the compiler (or, the IDE) will issue an error.

② An array object has the built-in length property, which returns `number`.

③ This object has an explicitly declared `length: number` property, and hence it can be used with this function.

④ This will output `10`.

Chapter 10. Arrays

10.1. Generic `Array<T>`

An array object in Javascript represents a sequence of items. In general, the types of the items in a given array need not be the same. When an array is homogeneous, that is, when every item in the array is the same type, `T`, Typescript uses the `T[]` syntax to denote the type of the array object. Alternatively, the parameterized type syntax can be used, e.g., `Array<T>`.

For example,

```
const abc: string[] = ['a', 'b', 'c'];   ①
const num: Array<number> = [];           ②
```

① The type annotation is not needed. `string[]` is the same as `Array<string>`.

② The type annotation is necessary since the type of the initial value is ambiguous. `Array<number>` is the same as `number[]`.

For heterogeneous arrays, that is, for the arrays with different type items, we can use the narrowest type that is compatible with all the item types. Or, in an extreme case, we can use an array of `any` or `unknown`. For instance,

```
const mix: any[] = ["one", 2, "three"];  ①
```

① The inferred type in this case would be `(string | number)[]`.

Note that `any[]` is not the same as `any`. `any[]`, or `Array<any>`, is an array type, whose item types are arbitrary. On the other hand, `any` is a type that does *not* carry any static type information.

10.1.1. Array creation

An array object, in Javascript and Typescript, can be created in a few different ways. First, the literal syntax:

```
const pets = ["dog", "cat"];            ①
```

① The inferred type of `pets` is `string[]`.

Alternatively, one can use the `Array` constructor function:

```
const empty = new Array<number>(10);    ①
empty[0] = 33;                          ②
// empty[4] = "awkward";                ③
// empty.push("smooth");                ④
```

① This creates a 10-element array of `number`, whose type is `number[]`. Without the generic type argument, the constructor will create an array of the `any[]` type.

② You can assign a `number` value to any of the array slots, or add any `number` values, e.g., using the `push` method.

③ This results in a compile-time type error.

④ Likewise, this is not allowed.

The `Array` constructor function is overloaded. Here's a different way to call the constructor:

```
const pets = new Array("dog", "cat");   ①
```

① In this case, the generic type argument is not really needed if the intended type of `pets` is `string[]`. That is, `new Array("dog", "cat")` is the same as `new Array<string>("dog", "cat")`.

An array can also be created based off another existing array. The simplest way is to use the array spread operator (`...`):

```
const petsCopy = [...pets];
```

Or, using the `Array.from` method,

```
const petsCopy2 = Array.from(pets);
```

Another common way is to copy an array, using the `slice` method,

```
const petsCopy3 = pets.slice();
```

10.1.2. Array iteration

Arrays, and other iterable objects like strings, can be iterated with the `for of` statement. For example,

```
for (const pet of pets) {                ①
    console.log(pet);
}
```

① The type of the loop variable is automatically determined by the iterating array. That is, the type of `pet` is `string` in this example since the type of `pets` is `string[]`. If you need to use a different type, then you can use the type assertion on `pets`. For instance,

```
for (const pet of pets as any[]) {       ①
    console.log(pet);
}
```

① The type of `pet` in this example is `any`.

One thing to note is that using a non-iterable object in the `for - of` statement in Javascript will cause a *run-time* exception. In Typescript, this kind of errors are caught *at compile time*.

10.2. Generic `ReadonlyArray<T>`

For variables referring to arrays for which no mutation is intended, we can declare them as `ReadonlyArray`. The `ReadonlyArray` type describes arrays that can only be read from. Any variable declared as `ReadonlyArray` cannot add, remove, or replace any elements of the array.

We can use the generic type syntax, `ReadonlyArray<T>` or the literal syntax `readonly T[]`. For example,

```
const safariPets: readonly string[]      ①
    = ["wolf", "lion", "rhino"];
const waterPets: ReadonlyArray<string>
    = ["goldfish", "angelfish"];
```

① Normally, the type inference will yield the `string[]` type, in this example, and hence an explicit type annotation is needed for readonly array types.

Readonly arrays can be created in the same way as the normal (non-readonly) arrays, except for different type annotations on the variables, and they can be iterated over just like normal arrays. For instance,

```
for (const pet of safariPets) {
    console.log(pet);
}
```

Chapter 11. Algebraic Data Types

There are a number of different ways to create a new type from existing types. We will discuss a few of the common such methods in this chapter, including tuple types and union types. Some other methods are discussed later in the book, e.g., in the Advanced Types chapter.

11.1. Tuples

11.1.1. Fixed-size tuples

Many programming languages support `tuple` types. Javascript doesn't (as of ES2023). Typescript adds the tuple support on top of Javascript. In general, a tuple is a sequence of *a fixed number of* elements, possibly with different types. The order is important.

Typescript uses a comma-separated list of elements enclosed in square brackets `[]` to denote a tuple literal. Likewise, it uses a comma-separated list of element types enclosed in square brackets `[]` to denote the type of a tuple.

For example,

```
type StrPair = [                    ①
  firstName: string,                ②
  lastName: string,
];                                  ③
```

① `StrPair` is an alias to a tuple type, `[firstName: string, lastName: string]`.

② Tuple elements can all be named, or they can all be anonymous, in a given tuple type. When elements are named, the tuple is called a labeled tuple.

③ This declaration is essentially equivalent to `type StrPair = [string, string]`.

```
const myName: StrPair = ["Harry", "Potter"];   ①
let grade: [string, number] = ["ID-1234", 55]; ②
// grade = myName;                             ③
```

① We use the type alias, `StrPair`, defined earlier to annotate the variable `myName`. Its initial value is consistent with the annotated tuple type.

② The type of `grade` is `[string, number]`, a two-element tuple.

③ The variables, `myName` and `grade`, have different types, and their values cannot be assigned to one another. This statement will cause a type checking error.

A tuple element can be accessed using the index notation. For instance, using the above example,

```
console.log(myName[1]);                        ①
console.log(`${grade[0]}: ${grade[1]}`);       ②
```

① This statement will print out *Potter*. The type of `myName[1]` is `string`.

② This will print out *ID-1234: 55*. The types of `grade[0]` and `grade[1]` are `string` and `number`, respectively.

It should be noted that, in Typescript, a tuple type with element types `T1`, `T2`, ... and `Tn` extends from an array type, `Array<T1 | T2 | ... Tn>`, whose element type is the union type of all element types of the given tuple type.

11.1.2. Generic tuples

Tuple types are implicitly generic just like array types are. For example, three-element tuple types like `[string, string, number]` and `[string, number, number]`, etc. may all be viewed as realizations of an implicit generic type `[T1, T2, T3]`.

Furthermore, we can still use the generic tuple-like syntax, when needed. For instance,

```
function first<T1, T2>(tuple: [T1, T2]): T1 {
    return tuple[0];
}
const f = first([10, "hi"]);              ①
console.log(typeof f);                    ②
```

① This function call is the same as `first<number, string>([10, "hi"])`.

② The type of `f` is `T1`, which is `number`.

11.1.3. Variadic tuples

Tuple types are generally used to model sequences with fixed lengths and specific element types in many different programming languages.

In Typescript, however, tuple types are much more flexible, for better or worse. We just discussed labeled tuple types. In addition, tuple elements can be made optional. Tuple types can even include the "rest" elements, in the leading, trailing, and in fact any positions. They are called the *variadic tuples* in Typescript, as in variadic functions which support a variable number of arguments.

In fact, Typescript tuple types are sometimes used to model function parameter lists. Tuple type's flexibility also comes from the fact that tuples in Typescript are eventually transpiled to arrays in Javascript.

As a little digression, let's review how array spreading and destructuring works in Javascript.

```
> const arr1 = [1, 2]
> const arr2 = ['a', 'b']
> const arrX = [...arr1, ...arr2]        ①
> arrX
[ 1, 2, 'a', 'b' ]
```

① We use the spread operator `...` to "spread" the elements of both `arr1` and `arr2` into `arrX`.

```
> const [_, ...rest] = arrX              ①
> rest
[ 2, 'a', 'b' ]
```

① The `...` operator is used to collect the "rest" of the elements of `arrX`. This is known as the "destructuring assignment". The first element, after destructuring, is discarded using the `_` variable in this example.

Typescript's variadic tuples use a similar syntax. For example,

```
type StrNums = [string, ...number[]];    ①
```

① This tuple type can have one or more elements, with the first element of the `string` type and the rest of the `number` type.

```
let t: StrNums = ["Hermione"];           ①
t = ["Ron", 100];
t = ["Dudley", 10, 21, 32, 43];
```

① Some of the valid values assignable to the variable `t`. The rest element can comprise zero, one, or more elements, of the `number` type in this example.

Here's another example,

```
type NumberPair = [number, number];
type Strings = string[];
type Structured = [...NumberPair, ...Strings, boolean];
```

Note that the *spread* operator syntax can only used with tuple or array types, and these element types do not have to be the last element. In this example, the type `Structured` is the same as `[number, number, …string[], boolean]`.

In addition, certain elements of a tuple type can be made optional, using the `?` suffix notation. For instance, a type `[string, number?]` is more or less comparable to a union type `[string] | [string, number]`. Or, a labeled tuple `[one: number, two?: boolean]` is roughly equivalent to `[one: number] | [one: number, two: boolean]`.

Here are some more examples.

```
let a: [number, boolean?, string?];      ①
a = [10];
a = [20, true];
a = [25, false, "one million"];
a = [25, undefined, "future"];           ②
```

① Only the trailing elements in a tuple type can be made optional. The type of an optional element automatically includes the `undefined` type. That is, this particular type is equivalent to `[number, (boolean | undefined)?, (string | undefined)?]`.

② Note that you cannot omit non-trailing elements. In this example, we just use `undefined` to "skip" the middle element.

11.2. Readonly Tuples

We can prefix any tuple type with the `readonly` keyword to make it a *readonly tuple*. Readonly tuples extend from `ReadonlyArray`. That is, a readonly tuple with element types `T1`, `T2`, … and `Tn` inherits from `ReadonlyArray< T1 | T2 | … Tn>`.

Unlike regular tuples whose slots could be written to, readonly tuples only permit reading from those positions. The `length` property of a readonly tuple is readonly as well, e.g., even when the tuple has trailing optional or rest element types.

```
type Grade = readonly [string, number];  ①
let g: Grade = ["Professor Dumbledore", 95];
console.log(`Name: ${g[0]}`);            ②
// g[1] = 75;                            ③
// g.push(30);                           ④
```

① A readonly tuple of two elements, `string` and `number`.

② We can access each element of the readonly tuple.

③ But, we cannot update the elements of readonly tuples.

④ The array's `push` method cannot be used with readonly tuples.

11.3. Union Types

A type essentially represents a (mathematical) set of all possible values belonging to the type. For example, a `string` type is a set of all possible string values, like `"hello"`, `"apple"`, and so on. Likewise, a `number` type is a set of all possible number values, like `5`, `10.5`, and so on.

Two or more types can be composed into a single new type by combining all their possible values. This is called a *union type*. Union types are sometimes called the sum types because they correspond (in

some abstract sense) to the algebraic sum of sets, or types. In contrast, tuple types are sometimes called the product types, again, because they correspond to the algebraic product of sets, or types. Tuple and union types are often called the algebraic data types for this reason.

Now we can define and use a union type of `string` and `number`, as follows:

```
let x: (string | number);                ①
x = "secret";                            ②
console.log(x);
x = 42;                                  ③
console.log(x);
```

① The `|` syntax is used to define a union type. Unlike the tuple types, the order is not significant in union types.

② A `string` value is of the type `string | number` (in addition to being of the type `string`).

③ A `number` value `42` is also of the type `string | number`, and hence it can be assigned to the variable `x`, whose declared type is `string | number`.

```
type TruthOrNumber = boolean | number;   ①
let u: TruthOrNumber = 42;
console.log(u);
u = true;
console.log(u);
```

① Type aliases are commonly used for union types because aliases are usually shorter and more meaningful, and they tend to be more easily reusable.

As for the operations allowed on the values of a union type, they must exist in every member type of the union. Otherwise, the operation may

not be applicable for certain values of the union type. For instance, while two numbers (of the `number` type) can be added, two values of a union type `number | boolean` may not be addable in general since one of the two values happens to be a `boolean`, and not a `number`.

11.3.1. Narrowing

A value that belongs to a union type also belongs to at least one of the member types of the union. For instance, using the previous example, while a value `100` belongs to the union type `boolean | number`, it is also a value of the `number` type.

We can perform operations on the values of a union type if they happen to belong to a more specific member type, through "narrowing". For example, using `type U = boolean | number`,

```
function addOrFail(a: U, b: U): U {
    // console.log(a + b);                ①
    if (typeof a == 'number' &&
        typeof b == 'number') {           ②
        return a + b;                     ③
    } else {
        return false;                     ④
    }
}
console.log(addOrFail(10, 20));           ⑤
console.log(addOrFail(10, true));         ⑥
```

① `a` and `b` cannot be added at this point since either one or both can be `true` or `false`. The compiler will throw an error.

② Using the "`typeof` type guard" is one way of type narrowing.

③ `a` and `b` can be any numbers like `100` or `2000`, or `true` or `false`, at run time, and yet the compiler knows at this point, at *compile time*, that these two values must be numbers, and they can be added. So, the compiler allows this statement.

④ This function returns a number or a boolean value, which *almost* reminds us of Javascript. But, note that it is still statically and strongly typed. The type happens to be a union type of `boolean` and `number`.

⑤ This will print `30`.

⑥ This will print `false`.

Narrowing is further discussed later in the book. As for assignability,

- The values of any member type of a union type can be assigned to variables of the union type.
- The reverse is not necessarily true.

11.4. Discriminated Unions

A special kind of unions whose members have the same properties but with different values are called the *discriminated unions*. Those members are called the union's *variants*. The compiler can discriminate variants based on the values of this common property, called the *discriminant*. For instance,

```
type OKResponse = { status: 200 | 201, payload: string };
type ClientError = { status: 400 | 404, error: Error };
type ServerError = { status: 500, error: Error };
type HttpResponse =                          ①
    | OKResponse                             ②
    | ClientError
    | ServerError;
```

① `HttpResponse` is a discriminated union because each variant has the same property, `status`, and their value ranges are mutually exclusive.

② The leading `|` is optional.

Here's an example usage of the `HttpResponse` type:

```
function handle(response: HttpResponse): string | Error |
undefined {
    switch (response.status) {
        case 200: case 201:
            return response.payload;      ①
        case 400: case 404:
            return response.error;        ②
        case 500:
            throw response.error;         ③
        default:
            return;                       ④
    }
}
```

① Note the narrowing in action. Not all variants of the `HttpResponse` type include the `payload` property, but in this case we can safely use that property in this particular `switch` branch.

② Ditto.

③ The return type of a function that does not return can be annotated with `never`. But, `never` is a subtype of all other types in Typescript, and hence `never` need not be explicitly specified when other types are used, as in this example.

④ Returning no value is equivalent to returning `undefined`.

11.5. Intersection Types

Another way to combine two or more existing types is to take a (mathematical) intersection among all the sets of the possible values of the member types. This is called the *intersection type*. A value of an intersection type belongs to all of its member types. On the flip side, an operation that is allowed in any of the members types is a valid operation for the values of the intersection type.

For example,

```
type Coord =                                ①
    & { lat: number, alt: number }          ②
    & { lon: number, alt: number };
const c: Coord = {                          ③
    lat: 40.7,
    lon: -73.9,
    alt: 10.0,
};
console.log(c.lat, c.lon, c.alt);           ④
```

① `Coord` is (an alias to) an intersection type of two object literal types.

② An intersection type is defined using the `&` operator.

③ It may seem a little counterintuitive, but a value of the intersection type `Coord` will need to include all three properties from all member types. That is, in this example, `Coord` is equivalent to `{ lat: number, lon: number, alt: number }` (and, not `{ alt: number }`).

④ You can, therefore, access any of the properties (or, methods) from any of the members of an intersection type, on the values of the intersection type.

In terms of assignability,

- The values of an intersection type can be assigned to variables of any of its member type.
- The reverse is not necessarily true.

Chapter 12. Function Types

12.1. Function Definitions

Functions in Javascript can be declared as statements or they can be introduced as expressions. In both cases, functions are declared with

- The keyword `function`, followed by
- An optional function name,
- A formal parameter list within a pair of parentheses `()`, and
- A function body enclosed in a curly braces block `{}`. The function body comprises a series of zero or more statements. For example,

```
> function add(a, b) {                    ①
...     return a + b
... }
> add instanceof Function                 ②
true
```

① A function declaration with name `add`, and formal parameters `a` and `a`. The function body includes one statement, `return a + b`.

② The `add` function is an instance of `Function`.

The following function does not have a name,

```
> function(a, b) {                        ①
...     return a + b
... }
```

① A function declaration without a name. Otherwise, this function declaration is equivalent to the previous example.

A function expression example,

```
> const forever42 = function() {          ①
...     return 42
... }
```

① An anonymous function is assigned to a variable `forever42`.

12.2. Arrow Function Definitions

An arrow function, or Lambda function, is an expression that declares a function in Javascript. It has the general form, *Arrow Function Parameters* `=>` *Arrow Function Body*. The arrow function parameters can be

- A single parameter, or
- A parenthesized list of zero, one, or more parameters.

The arrow function body can be

- A single expression (that can be used syntactically on the right hand side of an assignment statement), or
- A function body block enclosed in a pair of curly braces `{}`.

For example,

```
> const f1 = x => x * x                   ①
> const f2 = () => 82828282               ②
> const f3 = (x, y) => {                  ③
...     const h = Math.sqrt(x * x + y * y)
...     console.log(h)
...     return h
... }
```

① The arrow function, assigned to `f1`, has a single function parameter `x` and its body is a single expression `x * x`.

② `f2` has an empty parameter list, and it has a single constant expression `82828282` as its body expression.

③ `f3` has two parameters, `x` and `y`, and its function body is a block, which includes three statements.

12.3. Function Types

12.3.1. Function type expressions

Function types are syntactically similar to arrow function expressions, using the fat arrow `=>` operator. For example,

```
type Strumber = string | number;
function add(                                ①
    a: Strumber, b: Strumber,                ②
): Strumber {
    return <any>a + <any>b;                  ③
}
```

① A function declaration with name `add` and formal parameters `a` and `b`. The function body includes one statement, which returns the value of `a + b`.

② Both parameters and the return value are type-annotated with the same type, `Strumber`. Note that the type annotations on function parameters are generally required.

③ This is *just* an illustration. Type assertions are discussed earlier.

This `add` function takes two arguments of the `Strumber` type and it returns the value of the same type. Hence, the type of `add` is `(a: Strumber, b: Strumber) => Strumber`. The formal parameters in the function types are required, but their exact names are ignored.

Here's the same function, written as an arrow function,

```
const add = (a: Strumber, b: Strumber): Strumber =>
    <any>a + <any>b;
```

When the parameters are type-annotated, even a single parameter needs to be enclosed in parentheses `()`. For instance,

```
const mirror = (me: {}): {} => {          ①
    return me;
}
```

① An arrow function expression is assigned to a variable `mirror`. It takes a single argument of type `{}` and returns the same value (of the same type).

A type alias can be used to name a function type. For example,

```
type BinaryFunction = (a: number, b: number) => number;

function doMath(fn: BinaryFunction, a: number, b: number):
number {
    return fn(a, b);
}

const multiply = (a: number, b: number) => a * b;
const x = doMath(multiply, 2, 3);          ①
const y = doMath((a, b) => a + b, 5, 10);  ②
```

① The value of `x` is `6`.

② The value of `y` is `15`. Note that the arrow function, `(a, b) => a + b`, used as the first argument to the `doMath` function, does not need to be type-annotated since its type can be inferred from the context, e.g., from the function type of `doMath`.

12.3.2. Function return types

- A function which does not return a value, or whose return value will always be ignored, may be annotated with a return type `void`.
- A function that does not return a value but still includes a `return` statement (even without a return value) can use the `undefined` return type. Roughly speaking, `void` implicitly includes `undefined`.
- Likewise, a function that returns a value sometimes but doesn't at other times may be annotated with a union return type that includes `undefined`.
- A function that never returns, e.g., because it always throws an exception or it includes an infinite loop, has a return type of `never`.
- Note that, as a universal type, `any` can also be used as a function return type, which essentially includes `undefined`, `void`, `never`, and anything else.

12.3.3. Generic function alias

Generic functions can also be type-aliased. For example,

```
type SecretFunc<T> = (a: T) => 42;        ①
type Swap<T1, T2> = (a: T1, b: T2) => [T2, T1];
type LengthwiseFunc<T extends { length: number }> = (a: T) =>
number;
```

① Note that the generic type parameters are declared with the alias, and not with the function type.

12.3.4. The `Function` type

Every Javascript function is a `Function` object. Typescript has a type `Function` corresponding to the JS global object `Function` constructor. `Function` includes common properties like `bind`, `call`, and `apply`. It

also has the special property that allows the values of `Function` to be callable. These calls return a value of the `any` type.

```
const f: Function = (a: number) => 2 * a;
const r = f();                              ①
```

① The variable `f` is "callable" since it is annotated as `Function`. The inferred type of `r` is `any`. Note that this does not violate `noImplicitAny` since the return type of `f` is explicitly `any`. `f()` is called an untyped function call.

In Typescript, broad types like `Function` or `Object` are rarely used. Instead, the use of more specific function or object types is generally preferred.

12.4. Parameter List

The function parameters (within a pair of parentheses) can be

- Empty,
- A comma-separated list of one or more formal parameters, optionally followed by a comma,
- A comma-separated list of one or more formal parameters, a comma, and the rest parameter, or
- The rest parameter.

For example,

```
function f1() { }                           ①
function f2(a: T1, b: T2) { }               ②
function f3(a: T1, b: T2,) { }              ③
function f4(a: T1, b: T2, ...c: T3[]) { }   ④
function f5(...c: T3[]) { }                 ⑤
```

① An empty parameter list.

② A list of one or more parameters. `T1`, `T2`, and `T3` represent arbitrary types.

③ The trailing comma has been allowed since ES2017.

④ The rest parameter `...c` follows the formal parameter list.

⑤ The rest parameter `...c` only.

12.4.1. Parameter initializers

Each formal parameter of a Javascript function can include an initializer (e.g., a default value). This is one way to declare optional parameters in Typescript. For instance,

```
function f1(a: T1, b = 10) { }                  ①
function f2(a = 'girl', b: T2,) { }             ②
function f3(a: T1, b = "boy", ...c: T3[]) { }   ③
```

① The second parameter `b` has an initializer, and hence usually it need not be explicitly annotated. The `f1` function can be called with one or two arguments in Typescript. E.g., `f1(1)` or `f2(1, 2)`. For the former, `a = 1` and `b = 10` whereas for the latter, `a = 1` and `b = 2`.

② Unlike in many other languages, the optional parameters in Javascript functions need not be limited to the trailing portion of the parameter list. If you call `f2()` with no arguments, in this example, then `a` is assigned `'girl'` and `b` is `undefined`. In Typescript, however, this syntax has somewhat limited uses since all arguments for non-optional parameters need to be provided.

③ Parameters with initializers can be used even when a function includes the rest parameter. The rest parameter is implicitly optional. In Typescript, this particular function `f3` can be called with one, two, or more arguments (but, not with zero arguments, which is legal in Javascript).

12.5. Optional Parameters

In Javascript, a function parameter (or, a variable, in general) that is not assigned an explicit value has the value, `undefined`. Hence, you can omit some trailing arguments when calling a Javascript function. (That is, all function parameters are effectively optional in Javascript.) In Typescript, this is generally not allowed. A function should be called with the specified number of arguments. As we have seen in the previous section, however, the arguments for the optional parameters with default values can be omitted, as long as they are all in the trailing positions in a particular call.

In addition, we can also explicitly declare one or more *trailing* parameters as optional, using the question mark `?` suffix. For example,

```
function echo(msg?: string): string {    ①
    if (!msg) {                           ②
        return "Huh?";
    }
    return msg.toUpperCase();
}
```

① `msg` is an optional parameter, for which the `undefined` type is automatically union'ed to its annotated type. That is, in this example, the type of `msg` is really `string | undefined`.

② This conditional check also acts as a type guard since it handles the case of `msg` being `undefined`. without this, we couldn't have called the `toUpperCase` method on `string | undefined`.

The `echo` function can be called with zero or one argument. E.g.,

```
console.log(echo());
console.log(echo("Who's there?"));
```

This `echo` function signature is essentially the same as the following, using the default value-based optional parameters:

```
function echo(msg: (string | undefined) = undefined): string {
    // ...
}
```

Note that the explicitly declared optional parameters with `?` cannot include default value initializers. That is, mixing two optional parameter specifications is not allowed.

12.6. The Rest Parameter

Javascript supports variadic functions through the rest parameter syntax (`...`), which allows a function to accept an arbitrary number of arguments. As we have seen earlier, a function can include at most one rest parameter as its last parameter.

For example,

```
function sum(...rest: number[]):number { ①
    return rest.reduce(                  ②
        (a, b) => a + b,                 ③
        0,                               ④
    );
}
```

① The type of the rest parameter is an array type.

② Can the argument `rest` be `null` or `undefined`?

③ Type annotation is not needed.

④ The initial value. Refer to a Javascript reference for more info on the `Array.reduce` method.

```
const s0 = sum();                          ①
// const s1 = sum(undefined);              ②
const s2 = sum(1, 2, 3);                   ③
const s3 = sum(...[1, 2, 3]);              ④
```

① If we do not provide an argument, the default value is an empty array. This is the same both in Javascript and Typescript.

② In Javascript, you can pass `null` or `undefined`, which the current implementation of `sum` cannot handle well. On the other hand, in Typescript, the rest argument is explicitly annotated to be `number[]`, and hence you cannot use `null` or `undefined` to call this function.

③ The variable `s2` will be initialized with a number `6`.

④ An alternative way to call a variadic function, e.g., using the array spread operator. This syntax is more commonly used when you already have an array, or a variable referring to an array.

The type of the rest parameter should be an array (`Array<T>` or `T[]`) or a tuple. Array types are more commonly used, but here are some examples of tuple-type rest parameters.

```
function first(...items: [number, number?, number?]): number {
    return items[0];
}
```

This function can take one, two, or three arguments, but no others.

```
// console.log(first());                   ①
console.log(first(5));
console.log(first(5, 7, 9));
// console.log(first(5, 7, 9, 11));
```

① Commented-out code means they do not pass static type checking.

The rest parameter with exactly three `number` arguments,

```
function threeSum(...nums: [number, number, number]): number {
    return nums[0] + nums[1] + nums[2];
}
```

This `threeSum` function can be called with three numbers, no less and no more. In general, requiring a fixed number of arguments may not be very useful for the rest parameters. More commonly used are tuple types with optional element types such as the `first` function above or with the spread element types. For example,

```
function product(...input: [string, ...number[]]): void {
    const fac = input.slice(1) as number[];    ①
    const p = fac.reduce((a, b) => a * b, 1);  ②
    console.log(input[0], p);
}
```

① Note that the tuple type `[string, ...number[]]` is a subtype of `Array<string | number>`, and hence the type assertion is needed here to declare `fac` as `number[]`.

② Again, type annotation is not needed for this arrow function.

Note that this function is equivalent to the following, which seems a bit simpler and a bit more readable:

```
function product(month: string, ...factors: number[]): void {
    const p = factors.reduce((a, b) => a * b, 1);
    console.log(month, p);
}
```

You can call either function implementation in the same way, with the first string argument and zero or more trailing number arguments.

```
console.log(product("March"));
console.log(product("April", 1, 5, 7));
// console.log(product("May", "Huh?", 100));    ①
```

① Illegal.

12.7. Parameter Destructuring

The modern Javascript supports *destructuring assignment*, as we have briefly discussed earlier with respect to the tuple types. We can also use objects in destructuring assignment.

For example,

```
const obj = { a: 1, b: 2, c: 3 };          ①
const { a, b } = obj;                      ②
console.log(a, b);
const { a: A, b: B } = obj;                ③
console.log(A, B);
const { a: head, ...tail } = obj;          ④
console.log(head, tail);
```

① This object has three properties of its own, `a`, `b`, and `c`.

② After destructuring, the new variables `a` and `b` will have the values `1` and `2`, respectively.

③ We can also map the object's properties to variables of different names. In this example, the variables `A` and `B` will be initialized with `1` and `2`, respectively.

④ The "rest" syntax. In this example, `head` has value `1` (`obj.a`) and `tail` has a value, `{ b: 2, c: 3 }`.

In addition, function parameters of Javascript functions can be declared with the destructuring syntax. For example,

```
> function print({ name, grade }) {      ①
...     console.log(name, grade)
... }
```

① This function has one parameter, which is written in the destructuring syntax. When `print` is called with an object, it will be destructured, and the corresponding property values will be assigned to these destructured variables.

```
> const record = {name: "Tweedledum", grade: 35}
> print(record)                          ①
Tweedledum 35
```

① Since the object `record` has both `name` and `grade` properties, we can call `print` with `record`. If the function is called with an argument with an incorrect "structure", it will raise a run time error in Javascript.

```
> print({name: "Tweedledee", grade: 35}) ①
Tweedledee 35
```

① Although Javascript does not support the "named argument" syntax, it can be sort of emulated with this destructuring parameter syntax.

In Typescript, the function parameters need to be annotated (unless they can be contextually inferred). In this example, we can do this,

```
function print({ name, grade }:
    { name: string, grade: number }      ①
): void {
    console.log(name, grade);
}
```

① A destructuring parameter annotated with an object literal type.

Alternatively, we can use a type alias or an interface type,

```
type MyRecord = { name: string, grade: number };
function print({ name, grade }: MyRecord): void {
    console.log(name, grade);
}
```

12.8. The `this` Parameter

The behavior of Javascript's `this` operator can be rather confusing even to seasoned Javascript developers. The value of `this` is determined each time a (non-arrow) function is called, at run time, and hence it can be different every time. On the other hand, arrow functions do not provide their own `this` binding, and they use the `this` value from the enclosing lexical scope, if any.

12.8.1. Global context

In general, the semantics of `this` is context-dependent. In Javascript's non-strict mode, the global `this` refers to the `globalThis` object. For example, the global `this` object may be the `Window` object in a Web browser or something else in different runtime environments such as Node.js. For instance,

```
> function whatIsThis() {
...     console.log(this.toString())
... }
> whatIsThis()
[object global]                          ①
> this === globalThis
true
```

① The actual `this` object in the global context will be different, for instance, depending on whether you use a Web browser's developer console or Node.js REPL to run this script.

In contrast, in Javascript's `"use strict"` mode, `this` is always `undefined` in the global context.

12.8.2. Function context

In a regular function context, when a function is called on an object, `this` refers to the object that is called with this function. For instance,

```
> function whatsThis() {
...     "use strict"                        ①
...     console.log(this.toString())
... }
> const obj1 = { whatsThis, toString: () => "obj1" }
> obj1.whatsThis()                        ②
obj1
> const obj2 = { whatsThis, toString: () => "obj2" }
> obj2.whatsThis()                        ③
obj2
```

① In REPL, Javascript uses the non-strict mode by default (except in a class definition, etc.). We enable the strict mode here for illustration.

② In this context, `this` is `obj1`. The fact that this object's property, `whatsThis`, and its value (a variable or function name) have the same name is incidental, but it is a somewhat common practice in modern Javascript programming.

③ In this context, `this` is `obj2`.

In Typescript, first of all, the global `this` is always `undefined` since `"use strict"` is always implied in Typescript code. In a function or class context, the type of `this` will need to be explicitly type-annotated, with `noImplicitThis` set to `true`, which is the case in this book.

This is done through the Typescript-specific `this` parameter in the function declaration, as a first parameter. The purpose of `this` parameter is to allow type annotations. Otherwise, at run time, `this` follows the same Javascript semantics. For example,

```
function whatsThis(
    this: { whatsThis: () => void }      ①
) {
    console.log(this.toString());        ②
}
const obj1 = { whatsThis, toString: () => "obj1" };
obj1.whatsThis();                        ③
const obj2 = { whatsThis, toString: () => "obj2" };
obj2.whatsThis();                        ④
```

① The `this` parameter will need to be annotated.

② You can use `this` in accordance with the annotated type. In this particular example, we simply call the `toString` method, which is part of `Object.prototype`.

③ The `this` parameter does not exist at run time, and we do not pass in any values for `this`. At run time, `this` will be `obj1`. Note that we are allowed to call the `whatsThis` method on this object (in the static analysis) because the method's function declaration takes `this` parameter which include the `whatsThis` method property.

④ This will print out `obj2`.

12.9. Typescript Function Overloading

Unlike in Javascript, functions can be overloaded in Typescript. That is, one can define multiple functions (or, function signatures) with the same name. (In Javascript, only some of the builtin functions are overloaded such as the `Array` constructor, for instance.)

12.9.1. Overload signatures vs the implementation signature

When you overload a function in Typescript, you provide a set of function signatures (without implementations), called the overload signatures, and one implementation, whose function signature is called the implementation signature. The implementation signature is not accessible (e.g., not callable). All overload signatures must have different parameter lists, and the implementation signature should be "compatible" with all overload signatures.

An example is worth a thousand words:

```
function aspectRatio(                          ①
    screen: string
): number;
function aspectRatio(                          ②
    width: number, height: number
): number;
function aspectRatio(                          ③
    arg1: (string | number), arg2?: number
): number {
    if (typeof arg1 == "string") {             ④
        switch (arg1) {
            case "Standard": return 1.33;
            case "HD": case "FHD": return 1.78;
            default: return NaN;
        }
    } else {
        return arg1 / arg2!;                   ⑤
    }
}
```

① An overload signature with one `string` parameter.

② An overload signature with two `number` parameters.

③ An implementation signature with two parameters. Note that the second parameter is declared as optional. When the type of the first argument `arg1` is `string` and the second argument is omitted, this call is the same as calling the first overload signature. When both arguments `arg1` and `arg2` are provided and they are both of the `number` type, this call is the same as calling the second overload signature. Therefore, this implementation signature *covers* both overload signatures. Note that this is just one way, and there may be other implementation signatures that are *compatible* with these two overload signatures. Any function signature that can support all overload signatures can be used as an implementation signature.

④ In this particular implementation, this type guard distinguishes which overload signature has been used to call this overloaded function.

⑤ We use the non-null assertion in this `else` branch since `arg2`, which corresponds to the `height` parameter in the second overload signature, cannot be `null` or `undefined`. Note that the implementation signature cannot be directly called, and there are only two ways for this implementation to have been invoked, through either of the two overload signatures. Hence, the implementation does not have to take care of all possibilities based on its own implementation signature (which can be largely arbitrary, as indicated).

As you can see, function overloading in Typescript works rather differently than in most other programming languages that support function overloading. If you are new to Typescript, we suggest that you consider other alternatives first, such as using optional parameters or using union type parameters, before committing on function overloading, for any given problem.

Chapter 13. Object Types

Everything is an object in Javascript. "Object" is one of the most overloaded, and possibly abused, terms in programming, and what exactly this statement means will not be entirely clear unless we know what is the *object* in this sentence.

We have seen Typescript's predefined type `object` earlier. The `object` type is (sort of) a supertype of any type that is not a primitive type. Javascript has the `Object` constructor, which is at the top of the prototype chain for all other objects such as `Array` and `Function`. Typescript has the corresponding type `Object`.

Now, in the context of the present chapter, an object refers to anything that has, or can have, a property. And, that is indeed just about *everything*. Typescript allows defining various object types based on the object's "shapes", or "structures", e.g., the presence and absence of different properties.

13.1. Object Literal Types

Objects are one of the most important components in Javascript. Likewise, object types are one of the most fundamental constructs in Typescript.

We can declare an object type in a number of different ways. Interfaces and classes provide two formal ways to create object types. In this chapter, we will discuss another way, one of the simpler and more basic methods, called the object literal types. We have been using the object literal type syntax throughout this book. We can declare an object type by listing pairs of its property names and their types, separated by commas `,` or semicolons `;`, within angular brackets `{}`. This is, in fact, rather similar to the way an object literal is represented in Javascript. For example,

```
type Song = {
    composer: string;                    ①
    singer: string;                      ②
};
```

① Unlike object literals, the property separators can be either commas (`,`) or semicolons (`;`), or even mixed.

② The trailing separator is optional.

Note that all properties require explicit type annotations under `"noImplicitAny": true`. Here's another example:

```
const t1: { x: number } = { x: 1 };      ①
// const t2: { x: number } = { };        ②
// const t3: { x: number }
//     = { x: 1, y: 2, };                ③
// const t4: { x: number } = { x: "a" }; ④
```

① This object type, `{ x: number }`, could have been inferred.

② The object literal type specifies the structure of all values that belong to that type. In this case, the object `{}` lacks the property `x`, and hence it is not a value of the type `{ x: number }`.

③ In this case, the object `{ x: 1, y: 1 }` has an extra property `y`, and hence it is not a value of the type `{ x: number }` either.

④ A value `{ x: "a" }` is not a valid value of `{ x: number }` since the value of the property `x` is of the `string` type, not `number`. Hence this value cannot be assigned to `t4`, which is declared as a variable of type `{ x: number }`.

(Note: Refer to the structural subtyping section for additional explanation.)

13.1.1. The empty object literal type

The empty object type `{}` is a special notation. It is sort of a supertype of all types that have zero, one, or more properties. For example,

```
const t1: {} = { x: 1, };
const t2: {} = {};
const t3: {} = { x: 1, y: 2, };
```

The variables `t1`, `t2`, and `t3` from the previous example are all values of the broadest object literal type `{}`. This is even broader than the `object` type. The `{}` type can be used to annotate values of certain primitive types as well (that can be auto-boxed to `Object`).

```
const a: {} = 0_0_0;
const b: {} = BigInt(0_0_0);              ①
const c: {} = "0_0_0";
const d: {} = Symbol("0_0_0");
```

① The `BigInt` literal syntax, e.g., `0n` instead of `BigInt(0)`, can be used if the TS config *target* is set to `es2020` or later.

13.2. Object Type Members

Javascript objects can include a few different kinds of properties such as data properties and function properties, etc. Typescript object types support all corresponding property members, with some additions.

13.2.1. Optional properties

By adding a `?` after a property name, we can mark the property as optional, similar to the optional function parameters. For example,

```
type Geo = {
    name?: string;                          ①
    latitude: number;                       ②
    longitude: number;
    altitude?: number;
};
```

① The fields, `name` and `altitude`, are optional.

② On the other hand, `latitude` and `longitude` are required fields.

Similar to optional tuple elements and optional function parameters, the types of an object's optional properties are implicitly augmented with `undefined`. That is, the type of `name` in this example is effectively `string | undefined`. The type `Geo` with optional properties is more flexible and it can include a range of different object structures. E.g.,

```
const g1: Geo = { latitude: 10, longitude: 10 };
const g2: Geo = { name: "Starbucks", latitude: 20, longitude:
-20 };
const g3: Geo = { name: "ISS", latitude: 30, longitude: -30,
altitude: 1000 };

for (const g of [g1, g2, g3]) {             ①
    console.log(g.name?.length);            ②
}
```

① The modern Javascript supports iterations through `for - in` and `for - of` as well as through the classic C-style `for` loop.

② The use of the question mark `?` after `name` is called the optional chaining in ES2020. Typescript is always a bit ahead of ECMAScript standards, and it has been available in Typescript for some time. In this example, if `g.name` is `null` or `undefined`, it just returns `undefined`. Otherwise, it returns the value of `g.name.length`.

13.2.2. Readonly properties

Properties can also be marked as `readonly`. Readonly properties are similar to `const` variables. You cannot reassign different values to `readonly` properties. For instance,

```
type Stock = {
    readonly company: string,            ①
    price: number,                       ②
};
```

① `company` is a readonly property.

② `price` is a regular (non-readonly) property.

Here's an example usage:

```
const apple: Stock = { company: "AAPL", price: 1.0 };
apple.price = 2.0;                       ①
// apple.company = "MSFT";               ②
```

① This is fine.

② But, Typescript won't let us assign a different value to the readonly property, `company`.

Note that `readonly` properties are not truly immutable (just like `const` variables are not truly immutable). The value of a `readonly` property (or, `const` variable) can still change. For instance,

```
type Portfolio = {
    readonly stock1: Stock;              ①
};
```

① A `readonly` property, but not truly immutable.

```
const myPort: Portfolio = {
    stock1: { company: "GOOG", price: 10.0 },
};
console.log(myPort);
// myPort.stock1
//     = { company: "META", price: 0 };  ①
myPort.stock1.price = 1_000_000.0;       ②
console.log(myPort);
```

① We cannot assign a new value to the readonly `stock1` property.

② But, we can still modify its value.

13.3. Index Signatures

In Javascript, properties of an object can be accessed via the member access syntax (`.`). Alternatively, properties can also be accessed through the index notation (`[]`).

For example,

```
> const obj = { a: 1, b: 2 }
> obj.a
1
> obj["a"]                                  ①
1
> obj["b"] = 22                             ②
22
> obj
{ a: 1, b: 22 }
```

① The expression `obj["a"]` is equivalent to `obj.a`.

② The same with the write access.

As a matter of fact, the index notation provides a lot more flexibility.

```
> obj["c c"] = 33                          ①
33
> obj
{ a: 1, b: 22, 'c c': 33 }
```

① `c c` (with a space in the middle) is not a valid Javascript identifier, but it can still be used as a property key.

```
> obj[4] = 44                              ①
44
> obj
{ '4': 44, a: 1, b: 22, 'c c': 33 }
```

① Numbers are converted strings.

```
> obj["x" + 3] = 55                        ①
55
> obj
{ '4': 44, a: 1, b: 22, 'c c': 33, x3: 55 }
```

① An expression, which can be ultimately evaluated to `string`, can be used as an index, or property key.

13.3.1. Typescript index signatures

In Typescript, all object types define more or less *fixed shapes*. The index signature syntax, however, allows defining an object type with an essentially infinite number of properties, e.g., with a particular type for values of those properties. For instance,

```
type Characters = {
    [key: string]: string;                 ①
};
```

① This index signature allows the `Characters` type to be usable with any objects which have zero, one, or more properties of the `string` or `number` type (but, nothing else). (The `number` type is implicitly included because number keys are automatically converted to `string` in Javascript.)

Here's one example of an object that can be assigned to a variable of this `Characters` type:

```
const incredibles: Characters = {
    bob: "craig",                          ①
    helen: "holly",
};
```

① Although `Characters` does not include fields named `bob` or `helen`, this object is compatible with this type.

An object type with an index signature can include additional named properties or even other index signatures with different key types, from `number`, `string`, or `symbol`. However, all properties should be "consistent" with the index signatures. That is, all value types of the index signatures should be assignable to each other, and the types of any named properties should be assignable to all index signatures.

For instance,

```
type Movie = {
    [character: string]: string;           ①
    [year: number]: any;                   ②
    title: string;                         ③
//  price: number;                         ④
};
```

① The value type of this index signature is `string`.

② The value type of this additional index signature is `any`. This is legal since `any` and `string` types are assignable to each other.

③ An extra property with an explicit name `title`. Since its property type `string` is assignable to both `string` and `any`, this is legal.

④ The `number` type property is, however, not compatible with the first index signature, whose value type is `string`.

Here's one example usage:

```
const incredibles: Movie = {
    ['title']: "The Incredibles",        ①
    ['Bob Parr']: "Nelson",
    ['Helen Parr']: "Hunter",
    [2010]: "December",                  ②
};
```

① In Javascript, the property syntax, `title: "Incredibles"` vs `['title']: "Incredibles"`, have no difference. As we can easily see from this example, the named property is just a special case of more general index signatures specified in the `Movie` type. Note, however, that `title` is a required property of `Movie`.

② In Javascript, as indicated earlier, `number` type keys are converted to `string`. And hence there is no practical difference between the two index signatures defined in this example type, `Movie`.

13.4. Getters and Setters

Javascript supports `get` and `set` syntax, which can be used to provide some kind of "pseudo properties". Getter and setters are method properties, but they afford the data property access syntax. Common use cases are to expose a computed value as a property or to add a validation logic when a property value is set, etc.

In general, a getter and a setter are used together as a pair to represent a single pseudo-property. A getter without a setter implies a readonly property. In Typescript, the corresponding getter and setter can be annotated with different types as long as the getter type is assignable to the setter type.

Here's a simple example,

```
const ruler = {
    get size(): number {                ①
        return 12;
    },
    set size(value) {                   ②
        // Ignored
    }
};
console.log(ruler.size);
```

① A getter syntax, using the keyword `get`.

② A setter syntax, using the keyword `set`. The type of the setter parameter can be inferred from the corresponding getter. That is, in this example, the type of `value` is `number`.

Typescript treats a getter-setter pair as a single property for the purposes of typing. For example, the type of `ruler` is `{ size: number }`. In case of a getter without a matching setter, it is mapped to a `readonly` property.

A slightly more complex example:

```
const qubit = {
    angle: 0,           // In radians
    get phase() {                               ①
        return this.angle;
    },
```

```
    set phase(angle:
        | number
        | [number, "deg" | "rad"]
        | "zero") {                          ②
        if (angle == "zero") {               ③
            this.angle = 0;
        } else if (typeof angle == "number") { ④
            this.angle = angle;
        } else {                             ⑤
            const [a, unit] = angle;         ⑥
            switch (unit) {
                case "deg":
                    this.angle = a * Math.PI / 180;
                    break;
                default:
                    this.angle = a;
            }
        }
    },
};
```

① The type of the getter is inferred to be `number` based on the `this.angle` property.

② The value of `number` is assignable to the setter type, a union type of `number`, `[number, "deg" | "rad"]`, and `"zero"`, and hence this is valid although getter and setter have different types.

③ This value equality effectively acts as a type guard for the literal type, `"zero"`.

④ Another type guard example using the `typeof` operator. At this point, the type of the input argument `angle` is `number`.

⑤ In this `else` block, the type of `angle` must be the tuple type, `[number, "deg" | "rad"]`.

⑥ Typescript's tuple destructuring is the same as array destructuring in Javascript.

Here's a test code:

```
qubit.phase = Math.PI / 2;
console.log(qubit.phase);
qubit.phase = [45, "deg"];
console.log(qubit.phase);
qubit.phase = "zero";
console.log(qubit.phase);
```

The type of `qubit` is

```
{
    angle: number;
    phase: number | [number, "deg" | "rad"] | "zero";
}
```

13.5. Member Methods

Method properties can use a couple of different syntax. In addition, method properties can also be declared as optional. For instance,

```
type Pitcher = {
    ball(count: number): void;           ①
    strike: (count: number) => void;     ②
    cheer?(): string;                    ③
};
```

① A function syntax. The function return types are required even when they are `void`.

② A field with an arrow function property.

③ `cheer` is an optional method property of `Pitcher`.

Here's an example object that is compatible with, and assignable to, `Pitcher`.

```
const p1: Pitcher = {
    ball(count: number): void { },
    strike: (count: number) => void {},
};
```

13.5.1. Method overloading

Methods, declared with the function syntax, can be overloaded. That is, an object literal type can include multiple method members with the same name as long as they have different sets of function parameters. For instance,

```
type Keeper = {
    catch(ball: string): boolean;
    catch(flower: [string, number]): boolean;
};
```

Function overloading in Typescript is explained in the previous chapter on function types.

13.6. Structural Subtyping

Many strongly typed programming languages use a hierarchical type system. One type can be a subtype of another type, and one can be a supertype of another. Many such type systems often include one top-level base type, from which all other types in the system inherit.

The type system of Typescript works somewhat differently. With the exception of some primitive types, the relationships among the types are determined by their shapes, or structures.

For example,

```
let v1 = { a: 1 };                    ①
let v2 = { a: 1, b: 2 };              ②
v1 = v2;                              ③
// v2 = v1;                           ④
```

① The inferred type of `v1` is `{ a: number }`.

② The inferred type of `v2` is `{ a: number, b: number }`.

③ The types, `{ a: number }` and `{ a: number, b: number }`, have no relationships like one being a subtype of another, etc., and yet `v2` of type `{ a: number, b: number }` is assignable to `v1` of type `{ a: number }`.

④ The reverse does not hold true.

Even in the absence of explicit inheritance relationships, the two types in this example are related in Typescript. Structurally, `{ a: number }` is broader than `{ a: number, b: number }`, and hence assigning a value of `v2` to `v1` works, in a similar manner that a value of a first type can be assigned to a variable of a different second type as long as the first type is a subtype of the second type, e.g., in the programming languages with hierarchical type systems.

(Note that, in the beginning of this chapter, we showed some examples, in which only the assignments between the same types worked. That is an exception. When an object literal is directly used for initialization or assignment, the types involved have to exactly match. This rule is in place in Typescript to reduce errors due to some trivial mistakes like having typos in the object literal property names.)

Chapter 14. Interfaces

Interfaces are one of the few fundamental ways to create new types from scratch in Typescript, along with object literal types and classes. Interfaces are used to specify the "shape" of Javascript objects, and their behavior at run time, just like object literal types. Interfaces can be merged, extended, and implemented by other types.

14.1. Interface Types

An interface declaration creates an object type with a given name:

```
interface PointA {                          ①
    x: number;
    y: number;
}
```

① Syntactically, an interface declaration, in its most basic form, comprises the TS keyword `interface`, followed by an interface name and an object literal type. An `interface` can, therefore, syntactically include all member types of object literal types, as specified in the previous chapter.

Besides the initial name given in the declaration, other names can also be assigned to the interface through type aliasing.

```
type PointB = PointA;
```

As indicated earlier, Typescript uses the structure-based type system, and all object types with the same "shape" are the same type regardless of how they are declared or how they are named. For instance,

```
type PointC = { x: number; y: number; }; ①
```

① Now, `PointA`, `PointB`, and `PointC` all refer to the same type.

The difference between the object literal type declarations and the interface type declarations is not their end result, but rather the language support in how we create an object type. In case of object literal types, essentially we list all properties of an object. In case of interface types, one can start by specifying each property, but Typescript provides more high-level support like inheritance and what not. In addition, interface declarations are "open-ended", meaning that you can add additional properties after an interface has been defined.

```
interface PointA {                      ①
    x: number;
}
type PointB = PointA;                   ②
interface PointA {                      ③
    y: number;
}
```

① The type `PointA` has one property `x`.

② `PointB` is the same type with one property `x`.

③ We redeclare the `PointA` interface, which now includes *both* properties `x` and `y`. This is known as the declaration merging. Note that `PointB` is just a type alias, and hence it will end up being like the new `PointA` with two properties.

14.2. Extending Interfaces

Typescript provides a syntactic shortcut for creating a structural subtype, using an inheritance-like syntax. More specifically, a structural subtype of an `interface` can be created by "extending" that interface.

For example,

```
interface One {
    a: number;
    b: string;
}
interface Two extends One {                ①
    c: boolean;
}
```

① One interface can extend another interface.

```
interface Two {                            ①
    a: number;
    b: string;
    c: boolean;
}
```

① The resulting interface `Two` includes all properties of its "parent interface", `One`, and hence it is guaranteed to be a structural subtype of `One`.

Note that effectively the same thing can be achieved using intersection types. For example,

```
interface Delta {
    c: boolean;
}
type Two = One & Delta;                    ①
```

① Intersection works even when one or both of its members are object literal types.

Chapter 15. Classes

The Typescript `class` has some type-specific, and other syntactic, extensions to Javascript classes, which was first introduced in ES2015.

15.1. The ECMAScript Class

15.1.1. Class declaration

Javascript classes are (implicitly) based on *prototypes*, and they provide some additional features that are unique to classes. A class is a template for creating objects. The ES2015 `class` is an extension of the traditional constructor function syntax. In fact, classes are (a special kind of) functions in Javascript.

A new `class` can be declared as an anonymous or named expression or as a class declaration statement. In all three cases, the class declaration starts with the keyword `class`, followed by a class name (which is optional in case of anonymous class) and a class body enclosed in a pair of curly braces `{}`. (Note that a class body is always strict mode code even without the `"use strict"` directive.)

```
> class MyClass1 {                          ①
...     // class body
... }
> const MyClass2 = class {                  ②
...     // class body
... }
> const MyAlias = class MyClass3 {          ③
...     // class body
... }
```

① A class declaration statement.

② An anonymous class declaration expression, which is assigned to a variable `MyClass2`. This class can be referred to by `MyClass2`.

③ A named class declaration expression. This class can be referred to by the variable `MyAlias`, but not by the class name `MyClass3`.

```
> [typeof MyClass1, typeof MyAlias]
[ 'function', 'function' ]
> [MyClass1 instanceof Function, MyAlias instanceof Function]
[ true, true ]
> MyAlias
[class MyClass3]
> MyClass3
Uncaught ReferenceError: MyClass3 is not defined
```

15.1.2. Class body

The class body comprises zero or more of the following properties:

- Instance fields,
- `static` fields,
- Instance methods, including at most one constructor,
- `static` methods,
- `static` initialization blocks,
- Instance getters and setters, and
- `static` getters and setters.

All properties are *public* by default. The properties whose names start with a hash `#` are, on the other hand, *private*, and they cannot be accessed outside the class/object. The `static` keyword defines a `static` method or field for a class, or a `static` block. Static properties cannot be directly accessed on instances of the class. Instead, they are accessed on the class itself.

15.1.3. Fields

```
> class FieldDay {
...     iField1;                          ①
...     iField2 = true;                   ②
...     #piField1;                        ③
...     #piField2 = 100;                  ④
...     static sField1;                   ⑤
...     static sField2 = "Hi";            ⑥
...     static #psField1;                 ⑦
...     static #psField2 = "Hello";       ⑧
... }
```

① An instance field.

② Another instance field with an initializer.

③ A private instance field.

④ Another private instance field with an initializer.

⑤ A `static` field.

⑥ Another `static` field with an initializer.

⑦ A private `static` field.

⑧ Another private `static` field with an initializer.

15.1.4. Accessors

Getter and setter members can be declared using the `get` and `set` syntax, respectively. The `get` syntax binds a property to a function to be called when the value of that property is accessed. The `set` syntax binds a property to a function which will be called when the value of that property is set.

Syntactically, they are the same as those declared in object literals.

15.1.5. Methods

```
> class Method {
...     publicMethod() {}
...     #privateMethod() {}
...     static publicStaticMethod() {}
...     static #privateStaticMethod() {}
... }
```

15.1.6. Constructors

A `constructor` is a special instance method of a `class`, which is to be used for creating and initializing an object instance of that class.

```
> class Frog {
...     constructor(color) {
...         this.color = color          ①
...     }
... }
> const frog = new Frog("green")
```

① An instance field in Javascript can be declared in a constructor.

15.1.7. Static blocks

A static initialization block of a class includes statements that are to be evaluated during the class initialization.

```
> class Unpredictable {
...     static capricious
...     static {
...         this.capricious = Math.random()
...     }
... }
```

15.1.8. Inheritance

The `extends` keyword is used in a class declaration to create a class that includes the prototype of another class in its prototype chain.

```
> class Egg extends Chicken { }          ①
```

① `Egg` and all its instances are `instaceof Chicken`.

15.2. The Typescript Class

A `class` declaration also defines a type in Typescript.

```
class Hat { }
const hat: Hat = new Hat();              ①
```

① The type of a variable `hat` is `Hat`. Typescript can infer this type since the variable is initialized with the constructor call expression.

15.2.1. Constructors

Pure type declarations like object literal types and interfaces do not include constructors. The primary purpose of classes, on the other hand, is to construct one or more instances of them, and hence they often include constructors with specific implementations.

```
class Belt {
    size: number;                        ①
    constructor(size: number) {          ②
        this.size = size;
    }
}
const belt = new Belt(42);               ③
```

① The type of the field, `size`, could have been inferred from the constructor implementation, and hence it is optional.

② The `this` parameter in a constructor function is implicit, whose type is always the class/type being declared. Likewise, the return type of a constructor cannot be explicitly annotated.

③ The arguments that correspond to the constructor's parameters need to be provided to the `new` constructor call expression. The type of `belt` is `Belt`.

In Javascript, there can be no more than one constructor for a class. In Typescript, on the other hand, constructors can be overloaded, with a similar syntax to that of function overloading.

15.2.2. Member visibility

In Javascript, a member of a class is either public (default) or private (for names prefixed with a hash `#`). In Typescript, one can use three different kinds of modifiers, `public`, `private`, and `protected`, for the by-default public members. Javascript's private members are runtime-private. That is, their accessibility, or more precisely lack thereof, is enforced by Javascript runtimes. On the other hand, Typescript's visibility modifiers are purely compile-time constructs.

```
class Attire {
    public overcoat;                        ①
    protected shirt;                        ②
    private pants;                          ③
    #underwear;                             ④
    constructor(overcoat: string, shirt: string, pants:
string) {
        this.overcoat = overcoat;
        this.shirt = shirt;
        this.pants = pants;
        this.#underwear = "None";
    }
```

```
    toString = () => `${this.overcoat}:${this.shirt}:${
this.pants}:${this.#underwear}`;
}

const myAttire = new Attire("Burberry", "T-Shirt", "Shorts");
console.log(`${myAttire}`);
console.log(myAttire.overcoat);          ⑤
// console.log(myAttire.shirt);          ⑥
// console.log(myAttire.pants);
// console.log(myAttire.#underwear);
```

① A `public` instance field. The `public` modifier is optional. That is, all (static or non-static) properties of a class are public by default.

② A `protected` field, which can be accessed in a subclass.

③ A `private` field, not accessible outside the class/instance.

④ A truly private field.

⑤ A public property can be accessed from anywhere.

⑥ A protected property can be only accessed within the class or its subclasses. Typescript compiler will issue an error for this line. Likewise, Typescript compiler will issue an error when you try to access TS-private or JS-private members of a class.

Note that, if we had declared `myAttire` as `any`, for example, we would have been able to access the TS-private and TS-protected members by bypassing the static type checker. On the other hand, trying to access a JS-private field, e.g., `myAttire.#underwear`, outside the class/instance is both compile-time and run-time errors.

15.2.3. Parameter properties

The primary use of the constructors is often to initialize the instance fields of a class, as can be easily seen from the examples above. Typescript provides a convenience syntax to declare public instance fields and initialize them in the constructor. For example,

```
class Velocity {
    constructor(
        public x: number,                ①
        public y: number,
        public z: number,
    ) { }
}
```

① A constructor parameter declared this way, with a `public` modifier, is automatically set as an instance field with the same name.

```
const vel = new Velocity(0.1, 0.2, 0.5);
console.log(vel.x, vel.y, vel.z);        ①
```

① We can verify that the instance includes these (implicitly-created) properties and they are indeed publicly accessible.

The above `class` declaration is equivalent to the following:

```
class Velocity {
    x; y; z;
    constructor(x: number, y: number, z: number) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

15.3. Abstract Classes

A `class` can be declared as not implementable using the TS-specific keyword `abstract`. An `abstract class` can include `abstract` members as well. For example,

```
abstract class Soul {                    ①
    abstract price: number;              ②
    nameYourPrice() {
        console.log(`My soul is ${this.price} dollars.`);
    }
}
// const mySoul = new Soul();            ③
```

① `Soul` is `abstract`. Note that when `constructor` is not explicitly declared in a JS class (or a TS class, abstract or otherwise), an empty constructor is automatically provided by the runtime/compiler.

② `price` is an `abstract` field, and it does not require an initial value, or other implementations.

③ Abstract classes cannot be instantiated.

An `abstract class` can be extended. For instance,

```
class KittySoul extends Soul {           ①
    price: number;                       ②
    constructor(price: number) {
        super();                         ③
        this.price = price;              ④
    }
}
const kittySoul = new KittySoul(2.99);   ⑤
kittySoul.nameYourPrice();               ⑥
```

① `KittySoul` inherits from `Soul`, using the JS `extends` syntax. `KittySoul` is not abstract. A subclass inherits all public and protected properties from its base class.

② The super class's abstract property needs to be implemented.

③ The constructor of the base class, even the abstract one, needs to be explicitly called, `super()`, with appropriate arguments.

④ The value of the `abstract` field `price` of `Soul` is set here.

⑤ A `KittySoul` can be instantiated.

⑥ The inherited `Soul.nameYourPrice` method will use the `price` of `KittySoul`. This function call will print *My soul is 2.99 dollars.*

15.4. Implementing Interfaces

The Typescript `class` supports the *class* `implements` *interface* syntax as in many OOP programming languages. But, "implementing" interfaces by classes works differently in Typescript from the corresponding constructs in other languages. In particular, the `implements` declaration in Typescript is optional. For example,

```
interface Ball {                          ①
    deflated: boolean;
}
class Football implements Ball {          ②
    deflated: boolean = false;
    color: string = "brown";
}
class SoccerBall {                        ③
    deflated: boolean = true;
    readonly panels: number = 32;
}
```

① The interface `Ball` includes one public field, `deflated`, of the `boolean` type. Note that properties of interfaces and object literal types are always public, and they cannot be used with the class visibility modifiers.

② The class `Football` *implements* `Ball`. It indeed includes the same public field `deflated` with the same type from `Ball`, as required by this `implements` syntax.

③ `SoccerBall` also includes the `deflated` field.

As far as Typescript's structural typing is concerned, both `Football` and `Soccer` are structurally compatible with `Ball`. Typescript does not treat `Football` and `SoccerBall` any differently. For instance, the following function expects an argument of the type `Ball` and it returns `Ball | undefined`.

```
function kick(ball: Ball): Ball | undefined {
    if (!ball.deflated) {
        return ball;
    }
}
```

This function can be called with either `Football` or `SoccerBall`.

```
const football = new Football();
const soccerBall = new SoccerBall()
console.log(kick(football));              ①
console.log(kick(soccerBall));            ②
```

① This will print out *Football { deflated: false, color: 'brown' }*.

② This will print out *undefined*.

In fact, any object that is *compatible* with `Ball`, with or without type names, can be used with this function:

```
const dodgeBall = {
    deflated: true,
    duration: 1000,
} as const;                               ①
console.log(kick(dodgeBall));
```

① The inferred type of `dodgeBall` is `{ readonly deflated: true; readonly duration: 1000; }`. The `const` assertion is discussed earlier in the book.

An exception is object literal values defined at the point of use. E.g.,

```
// kick({ deflated: true, price: 1.0, }) ①
```

① This will cause a compile time error since the given object is not the same as `Ball`. (It has an extra property.)

In general, the primary use of the `implements` declaration is to make sure that a given class conforms to the specified interface(s). If you state that a class `implements` an interface, and if you do not provide all necessary implementations of the interface's properties, Typescript will catch the error through static type checking.

15.4.1. Implementing multiple interfaces

A `class` can *extend* no more than one direct base class. But, a class can *implement* one or more interfaces. For example,

```
interface Mammal {
    legs: 2 | 4;
}
interface Flyer {
    fly(): void;
}
class Bat implements Mammal, Flyer {
    legs: 2 | 4 = 2;
    wings = 2;
    fly() { }
}
```

Note that the type of `Bat` is structurally narrower than, or it is a subtype of, `Mammal` since it contains extra properties, `wings` and `fly`. It is also structurally narrower than `Flyer` since it contains extra properties, `legs` and `fly`. Now, `Bat` can be used in places where `Mammal` or `Flyer` is expected.

```
const bat1: Mammal = new Bat();
const bat2: Flyer = new Bat();
```

15.5. Generic Classes

A Typescript `class` can also be defined generically.

```
class Chest<T extends {}> {                ①
    private treasure;
    constructor(treasure: T) {
        this.treasure = treasure;
    }
    get content(): string {                ②
        return this.treasure.toString();
    }
}
```

① This type constraint is needed to be able to call the `Object.toString` method on the generic type value.

② The (readonly) getter property syntax.

```
const chest = new Chest(42);                    ①
console.log(chest.content);                     ②
type TreasureChest = Chest<{ value: number }>;  ③
const treasure = new Chest({ value: 100 });     ④
```

① The types of both the variable and the type parameter can be inferred, and hence they need not be explicitly specified. This statement is the same as `const chest: Chest = new Chest<number>(42);`.

② We cannot access `chest`'s private field `treasure`. But, we can access it through the `content` pseudo-property.

③ We can also create a type alias from a generic type using specific type arguments.

④ This statement is equivalent to `const treasure: TreasureChest = new Chest<{ value: number }>({ value: 100 })`.

A generic class can also implement generic interfaces.

```
interface Safe<T> {
    asset: T;
}

class SafeHouse<T> implements Safe<T> {
    constructor(public asset: T) { }           ①
}

const safe = new SafeHouse("Jason Bourne");    ②
const house = new SafeHouse(1_000_000_000);    ③
```

① By using the constructor parameter property syntax in this example, we satisfy the `implements` requirement.

② The type of `safe` is `Safe<string>`, or `SafeHouse<string>`. Or, simply `{ asset: string }`.

③ Likewise, the type of `safe` is `Safe<number>`, etc.

Chapter 16. Type Narrowing

As we discuss throughout this book, Typescript can infer types for certain variables and functions. In fact, it can go further. In certain situations, Typescript can deduce the type of a variable to be more specific or narrower than explicitly annotated, or initially inferred. This is called the *type narrowing*.

16.1. Control Flow Analysis

Typescript can analyze code and decide which branches are reachable and which are not, for instance. This analysis of code based on reachability is called *control flow analysis*. TypeScript uses this flow analysis to narrow types as it encounters type guards and assignments.

16.2. The `typeof` Type Guard

As indicated, the `typeof` operator can be used to get the run-time type of a given value, which more or less represents one of the eight fundamental types in Javascript (with the exception of `null` and functions).

Typescript also uses any `typeof` expressions for static flow analysis, e.g., to narrow types. In this context, it is called the `typeof` type guard.

We have seen some examples throughout this book. Here's another example:

```
function callOrDie(
    f: string | Function, ...args: any[]       ①
): void {
    if (typeof f == 'function') {              ②
        const result = f(...args);             ③
        console.log(`Call result: ${result}`);
```

```
    } else {
        console.log("Nothing to call");
    }
}
callOrDie("hi");                                    ④
callOrDie((a: number) => a, 42);
callOrDie((a: number, b: number) => a + b, 1, 2);
```

① Calling `f` when `f` is not callable will result in a *catastrophic* run time error, and Typescript will not let you do it. ☺

② But, with this `typeof` type guard...

③ Now, it is safe to call `f` at this point. Note, however, that using the broad `Function` type is not generally considered type-safe.

④ A few examples of calling this `callOrDie` function.

16.3. The `instanceof` Type Guard

Javascript's `instanceof` operator can also be used as a type guard, which checks the prototype chain of a given object.

You can use the `instanceof` type guard with Javascript's builtin objects such as `Date` and `Error`, or any objects that are constructed with `new`. More specifically, as we have seen earlier, the types based on classes use their own prototypes, and the `instanceof` type guard can be useful to check their types.

For instance, with the following example types:

```
class Classy {
    affirmation = "I'm classy!";
}
class Sassy {
    walkAndTalk = () => "Sassy, sassy";
}
```

```
type Saucy = undefined;
```

```
function sayWhat(
    me: Classy | Sassy | Saucy,          ①
): string {
    if (me instanceof Classy) {          ②
        return me.affirmation;           ③
    } else if (me instanceof Sassy) {    ④
        return me.walkAndTalk();         ⑤
    }
    return "Huh?";
}
```

① The three members of this union type have pretty much no commonalities. Note that a value of a union type will contain the common properties that belong to *all* its member types.

② An `instanceof` type guard.

③ Typescript knows that, in this `if` branch, `me` is of the `Classy` type and it has the property, `affirmation`.

④ Another `instanceof` type guard.

⑤ Another example of Typescript's static flow analysis in action.

16.4. The `in` Operator Narrowing

Because Typescript is structurally typed, one of the best ways to check whether a specific property exists in a given object is to use the `in` operator.

In fact, this is one of the most widely used methods to check the same in Javascript at run time. Typescript's type narrowing using the `in` operator emulates this runtime behavior. For instance,

```
type Fish =
    | { weight: number; }
    | { weight: number; taste: string };

function taste(fish: Fish): string {      ①
    if ("taste" in fish) {                ②
        return fish.taste;                ③
    }
    return "meh";
}
```

① The `Fish` type may, or not may not, have a property `taste`.

② Type narrowing via the `in` operator.

③ This is safe to do at runtime, and hence Typescript allows it as well. In a more general context in which this pattern is commonly used in Javascript, Typescript may not be able to infer the exact type of the property. In such a case, the type of that field will be `unknown`, or the union of all types that are allowed. One can use the type assertion to narrow the type further.

Here's an example use of this `taste` function:

```
let angelfish = { weight: 100 };
console.log(taste(angelfish));

let goldfish = { weight: 10, taste: "delicious" };
console.log(taste(goldfish));
```

16.5. Discriminated Unions

As we have seen earlier, discriminated unions can be rather useful in many situations, and their *discriminant* properties can be used for the purposes of type narrowing.

Here's another example:

```
interface Circle {
    readonly kind: "circle";
    radius: number;
}

interface Rectangle {
    readonly kind: "rectangle";
    width: number, height: number;
}

interface Triangle {
    readonly kind: "triangle";
    base: number, height: number;
}

type Shape = Circle | Rectangle | Triangle;    ①
```

① The type `Shape` is a discriminated union since all its member types have the common property `kind` with different values.

```
function area(shape: Shape): number {
    switch (shape.kind) {
    case "circle":
        return Math.PI * shape.radius ** 2;    ①
    case "rectangle":
        return shape.width * shape.height;
    case "triangle":
        return 0.5 * shape.base * shape.height;
    default:                                   ②
        return 0;
    }
}
```

① Type narrowing in action.

② The type of `shape` at this point is `never` since we exhausted all possible variants of `Shape`.

Note that these types `Circle`, `Rectangle`, and `Triangle` could have been declared in any different ways, e.g., using object literal types or classes. The implementation of the `area` function only depends on the fact that `Shape` is a discriminated union.

If we use classes for these member types, we could also use the `instanceof` type guards. For instance,

```
interface Shape {
    readonly kind: string;
}

class Circle implements Shape {
    constructor(
        public readonly kind: "circle",
        public radius: number
    ) { }
}

class Rectangle implements Shape {
    constructor(
        public readonly kind: "rectangle",
        public width: number,
        public height: number,
    ) { }
}

class Triangle implements Shape {
    constructor(
        public readonly kind: "triangle",
        public base: number,
        public height: number,
    ) { }
}
```

Then, we can implement our `area` function as follows:

```
function area(shape: Shape): number {
    if (shape instanceof Circle) {
        return Math.PI * shape.radius ** 2;
    } else if (shape instanceof Rectangle) {
        return shape.width * shape.height;
    } else if (shape instanceof Triangle) {
        return 0.5 * shape.base * shape.height;
    } else {
        return 0;
    }
}
```

Note that this implementation is more *OOP-ish*, if that is a word. (In fact, we could have even implemented a different `area` method for each class.) This is very nice because Javascript's `class` has runtime support which corresponds to Typescript's compile-time types. On the other hand, discriminated unions are a more *functional style*, and they are also very powerful tools.

As we have alluded a number of times throughout this book, there are many different ways to achieve the same thing in Typescript. The choice is yours. But, be reminded that our goal is to ultimately generate a simple, efficient, and less error-prone Javascript code.

Chapter 17. Advanced Types

Typescript allows expressing types using other existing types and values. For instance, the type of an object, or the type of a particular key of an object, can be referred to with the `typeof` and `keyof` operators, respectively. Furthermore, a new type can be conditionally created based on other type expressions, etc. We will go through some of the ways in which a type can be denoted in terms of other types and values. We have discussed abstract data types and generics in earlier chapters.

It should be note that these features are primarily used for annotating complex Javascript code or for building special libraries, and they are not commonly used in everyday Typescript programming.

17.1. Template Literal Types

The template literal type provides a way to create special union types as a generalization of string literal types. It uses the modern Javascript's template string syntax (``). For example,

```
type name = "Jack-Jack Parr";
type Hero = `Super Baby: ${name}`;              ①
const t: Hero = "Super Baby: Jack-Jack Parr";  ②
```

① The `Hero` type is just a string literal type, `"Super Baby: Jack-Jack Parr"`, although it uses the template literal type syntax.

② An example usage.

Or, a bit more realistic example:

```
type Lang = "en" | "es";
type Country = "US" | "MX";
type LangCode = `${Lang}_${Country}`;
```

The type LangCode is the same as the following union type:

```
type LangCode =
    | "en_US"
    | "en_MX"
    | "es_US"
    | "es_MX";
```

17.2. The typeof Type Operator

As we have discussed in the previous chapter, Javascript's typeof operator can be used as a type guard. Moreover, when the typeof operator is used in the type context, with a variable or property operand, it refers to the type of the given operand. For example,

```
const greeting1 = "Hello";                ①
let greeting2: typeof greeting1;          ②
greeting2 = greeting1;                    ③
```

① The type of greeting1 will be inferred to be a literal type, "Hello".

② Hence, the variable greeting2 has the same type.

③ This assignment will always succeed regardless of the actual type of greeting1.

Note that this is a purely compile-time construct, and the typeof operator in this context behaves differently from Javascript's typeof operator, which returns the eight predefined string values at run time. Here's another example:

```
const item1 = {                           ①
    product: "Robot",
    price: 9.99,
};
```

```
type Item = typeof item1;                    ②
let item2: Item, item3: Item;                ③
item2 = {                                    ④
    product: "Quantum Laptop",
    price: 1_000_000_000,
};
```

① The type of `item1` is `{ product: string, price: number }`.

② This creates a type alias, `Item`, for this type.

③ Now we can use the type alias just like any other types.

④ This assignment works since the type of `item2` is `Item`.

17.3. The `keyof` Type Operator

Typescript's `keyof` operator takes an object type and it returns a union type of all of its keys. For example,

```
const shift = { left: 1.0, right: 2.0 };
type Key = keyof typeof shift;               ①
const key1: Key = "left";                    ②
// const key2: Key = "Right";                ③
```

① The type alias `Key` refers to a union type `"left" | "right"`;

② The value `"left"` is a valid value of the type `Key`.

③ A static type error.

In case of types with an index signature,

```
type Likes = { [k: string]: boolean };       ①
const post1: keyof Likes = "ID: 1201";       ②
const post2: keyof Likes = 1202;             ③
// const post3: keyof Likes = true;          ④
```

① The key type of this index signature is `string | number`. This is because the `number` keys of Javascript objects are automatically converted to `string`, as indicated earlier.

② Consistent.

③ Consistent.

④ Not consistent. A type error.

17.4. Indexed Access Types

We can use the indexed access syntax to refer to the type of the value of a specific property of a given object type. For example,

```
type Point = { x: number, y: string };

type X = Point["x"];                    ①
type Y = Point["y"];                    ②

const x: X = 1.0;                       ③
const y: Y = "2 feet";
```

① The type `X` is `number`.

② The type `Y` is `string`.

③ We can also use the indexed access type syntax directly. E.g., `const x: Point["x"] = 1.0`.

17.5. Conditional Types

Conditional types are those that can have one of two different types depending on other types. Syntactically, conditional types are similar to Javascript's ternary (conditional) expression using the generic type constraint syntax, namely, `T extends U ? A : B`. For example,

```
interface Mammal { name: string }

interface Man extends Mammal { legs: 2, arms: 2 };
interface Dog extends Mammal { legs: 4 };
interface Cat extends Mammal { lives: 9 };

type Quadped<T> = T extends { legs: 4 } ? T : never; ①
type Pet = Quadped<Man | Dog | Cat>;           ②
let pet: Pet;                                  ③
// pet = { name: "Man", arms: 2, legs: 2 };    ④
pet = { name: "Dog", legs: 4};                 ⑤
// pet = { name: "Cat", lives: 9};             ⑥
```

① A generic type alias for a conditional type.

② This will end up filtering out `Man` and `Cat` since they do not have a property with type `legs: 4`.

③ A variable annotated as `Pet`.

④ A `Man` is not a `Pet`.

⑤ A `Dog` is a valid `Pet`.

⑥ A `Cat` is not a `Pet`.

Here's another example:

```
type NumberHolder = { data: number };
type StringHolder = { text: string };
type BooleanHolder = { flag: boolean };

type Holder<T extends number | string | boolean>
    = T extends number                    ①
    ? NumberHolder
    : T extends string
    ? StringHolder
    : BooleanHolder;
```

① A nested conditional type declaration. The ternary operator is left-to-right associative.

17.6. Mapped Types

Mapped types can be created by changing the structure of an input type using a mapping-like syntax. In particular, a mapped type syntactically iterates over the property keys of a given input type. For instance,

```
type Poynt = { x: number; y: string; };  ①
type Point = {
    [Prop in keyof Poynt]: number;       ②
};
const y: Point = { x: 3, y: -3 };        ③
```

① Not a real `Point`.

② The mapped type syntax. The resulting type is `{ x: number; y: number; }`;

③ An example variable annotated with a real `Point`.

Mapped types are usually defined generically. Here's an example:

```
type Funcfy<T> = {
    readonly [P in keyof T]: () => T[P]; ①
};
type FuncPoynt = Funcfy<Poynt>;          ②
```

① The special prefix, `readonly` or `-readonly`, adds or removes the `readonly` modifier from each key, respectively. Note that we use the indexed access type syntax, `T[P]`, with the generic type parameter.

② The type of `FuncPoynt` is `{ readonly x: () => number; readonly y: () => string; }`.

A. How to Use This Book

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> — Benjamin Franklin

The books in this "Mini Reference" series are written for a wide audience. It means that some readers will find this particular book "too easy" and some readers will find this book "too difficult", depending on their prior experience related to programming. That's quite all right. Different readers will get different things out of this book. At the end of the day, learning is a skill, which we all can learn to get better at. Here are some quick pointers in case you need some advice.

First of all, books like this are bound to have some errors, and some typos. We go through multiple revisions, and every time we do that there is a finite chance to introduce new errors. We know that some people have strong opinions on this, but you should get over it. Even after spending millions of dollars, a rocket launch can go wrong. All non-trivial software have some amount of bugs.

Although it's a cliche, there are two kinds of people in this world. Some see a "glass half full". Some see a "glass half empty". *This book has a lot to offer*. As a general note, we encourage the readers to view the world as "half full" rather than to focus too much on negative things. *Despite* some (small) possible errors, and formatting issues, you will get *a lot* out of this book if you have the right attitude.

There is this book called *Algorithms to Live By*, which came out several years ago, and it became an instant best seller. There are now many similar books, copycats, published since then. The book is written for "laypeople", and illustrate how computer science concepts like specific algorithms can be useful in everyday life.

Inspired by this, we have some concrete suggestions on how to best read this book. This is *one* suggestion which you can take into account while using this book. As stated, ultimately, whatever works for you is the best way for you.

Most of the readers reading this book should be familiar with some basic algorithm concepts. When you do a graph search, there are two major ways to traverse all the nodes in a graph. One is called the "depth first search", and the other is called the "breadth first search". At the risk of oversimplifying, when you read a tutorial style book, you go through the book from beginning to end. Note that the book content is generally organized in a tree structure. There are chapters, and each chapter includes sections, and so forth. Reading a book sequentially often corresponds to the *depth first traversal*.

On the other hand, for reference-style books like this one, which are written to cover broad and wide range of topics, and which have many interdependencies among the topics, it is often best to adopt the *breadth first traversal*.

This advice should be especially useful to new-comers to the language. The core concepts of any (non-trivial) programming language are all interconnected. That's the way it is. When you read an earlier part of the book, which may depend on the concepts explained later in the book, you can either ignore the things you don't understand and move on, or you can flip through the book to go back and forth. It's up to you. One thing you don't want to do is to get stuck in one place, and be frustrated and feel resentful (toward the book).

The best way to read books like this one is through "multiple passes", again using a programming jargon. The first time, you only try to get the high-level concepts. At each iteration, you try to get more and more details. It is really up to you, and only you can tell, as to how many passes would be required to get much of what this book has to offer.

Again, *good luck!*

Index

@

`"allowJs": false`, 25
`"alwaysStrict": true`, 26
`"commonjs"`, 25
`"es2016"`, 24
`"es2020"`, 25
`"nodenext"`, 25
`"noImplicitAny": true`, 28
`"noImplicitThis": true`, 28
"rest" elements, 82
`"strict": true`, 27-28
`"strictNullChecks": true`, 27
`"use strict"`, 22, 29, 105
`"use strict"` directive, 126
`"use strict"` mode, 26, 105
`&` operator, 90
--strict flag, 27
`-readonly`, 152
.gitignore file, 26
64 bit floating point, 44
64 bit integer, 44
`?`, 111
`?` suffix notation, 84
_ variable, 83
`{}` type, 111
`|` syntax, 86
`...` operator, 83

A

`abstract`, 134
abstract, 134
`abstract class`, 133-134
Abstract Classes, 133
Abstract classes, 134
abstract data types, 147
`abstract` field, 134-135
`abstract` members, 133
abstract property, 134
accessibility, 131
Accessors, 128
algebraic data types, 86
algebraic product, 86
algebraic sum, 86
alias, 58-59, 80
aliases, 35, 86
`allowJs`, 25
`alwaysStrict`, 26
Angular, 19
angular bracket `<>`, 64
angular bracket assertion, 66
angular bracket syntax, 65
angular brackets, 72
angular brackets `<>`, 71
angular brackets `{}`, 59, 109
annotated tuple type, 81
annotated type, 98, 106
annotated types, 62
anonymous, 126
anonymous class declaration, 127
anonymous function, 32, 92
anonymous function

declaration, 72
`any`, 28, 38, 46-48, 52, 76, 79, 95
`any` Type, 46
`any` type, 46-47
`any` value, 47
`any` values, 46
`any[]`, 76
app development process, 20
argument, 100
arguments, 62, 100
Array, 55
`Array`, 109
array, 53-54, 56, 76, 78
`Array` constructor function, 54, 77
Array creation, 77
array destructuring, 119
Array iteration, 78
array literal syntax, 54
array literal value, 65
Array literals, 65
array object, 75-77
`Array` objects, 54
array operations, 54
array spread operator, 78, 100
array spread syntax, 74
array spreading, 83
array type, 55-56, 72, 76, 81, 99
Array Types, 54
array types, 55, 82, 84
array-like, 55
`Array.from` method, 78
`Array.reduce` method, 99
Arrays, 54-55, 78
arrays, 54-56, 79
arrays in Javascript, 82
arrow function, 63, 74, 92-94, 101
arrow function argument, 63
arrow function body, 92
Arrow Function Definitions, 92
arrow function expression, 94
arrow function parameters, 92
arrow function property, 120
arrow functions, 104
`as` clause, 35
`as` keyword, 35
`as` operator, 64
asserting, 48
assignability, 51-52, 88, 90
assignable, 61, 122
assignment, 122, 148-149
assignments, 122, 140
auto-boxing, 40

B

Babel transpilation, 19
base class, 134, 137
`BigInt`, 44
`bigint`, 44
`BigInt` for `bigint`, 40
`bigint` Type, 44
block, 59, 93
`boolean`, 44-45
`Boolean` for `boolean`, 40
`boolean` type, 44
broader type, 52
broadest type, 47, 55
build steps, 19
build tools, 20

build-time construct, 68
builtin functions, 40
builtin global object, 52
builtin objects, 40, 141

C

callable, 141
class, 126-127, 129, 131-132, 137, 146
`class`, 129, 133, 137, 146
Class body, 127
class body, 126-127
Class declaration, 126
class declaration, 33, 126
`class` declaration, 130, 133
class declaration statement, 126
class definitions, 26
class initialization, 129
class name, 126-127
class visibility modifiers, 135
Classes, 73
classes, 109, 123, 126, 145
code documentation, 16, 58
collection, 54
collection types, 55
colon `:`, 16
common property, 144
CommonJS, 29
commonjs, 25
compatibility, 24
compatible, 76
compatible, 136
compilation, 18
compile time, 47, 64
compile time error, 15
compile time error, 71, 137
compile-time, 132
compile-time construct, 58, 148
compile-time constructs, 131
compile-time error, 37-38
compile-time type error, 77
compile-time types, 146
compiled languages, 18
compiler, 15-16, 21, 28, 64, 67, 87-88
compiler settings, 21-22
`compilerOptions`, 24
`compilerOptions` options, 24
computed value, 117
conditional type, 151
Conditional Types, 150
Conditional types, 150
conditional types, 150
configuration file, 21
`const`, 36-37, 60
`const` assertion, 65-66, 136
Const Assertions, 65
`const` assertions, 65-66
`const` Declaration, 36
`const` declaration, 36-37
`const` variable, 36
`const` variables, 39, 60, 113
constant expression, 93
constraints, 74
constructor, 77, 127, 129, 131-132, 134
`constructor`, 129, 134
constructor call, 130
constructor function, 40, 42, 131
constructor function syntax, 126
constructor implementation, 131

constructor parameter, 133
constructor prototype chain, 55
constructor syntax, 44
Constructors, 129-130
constructors, 130, 132
constructor's parameters, 131
Control Flow Analysis, 140
control flow analysis, 140
curly braces block `{}`, 91
curly braces `{}`, 92, 126
current directory, 34
current folder, 21
custom type, 69

D

data properties, 111
data structure, 69
data types, 70
declaration, 123
Declaration `export`, 31
declaration merging, 124
declaration syntax, 36
declarations, 39, 124
declared types, 17
Default class `export`, 33
default export, 30, 32, 34
`default` export, 32
`default export` declaration, 32
Default exports, 32
`default` exports, 30
Default function `export`, 32
Default `import`, 34
default initial value, 38
default value, 100
default value initializers, 99
deployment, 19
deployment process, 19
destructured, 103
destructured variables, 103
destructuring, 83, 102
destructuring assignment, 83, 102
destructuring assignment, 102
destructuring parameter, 104
destructuring parameter syntax, 103
destructuring syntax, 102-103
dev dependency, 20
developer console, 105
developer tools, 16
development, 18-19, 22, 46
Development Process, 18
development process, 13, 18-19
development workflow, 19
discriminant, 88
discriminant properties, 143
discriminated union, 88, 144-145
Discriminated Unions, 88, 143
discriminated unions, 88
discriminated unions, 143, 146
dynamic imports, 33
Dynamic languages, 18
dynamically typed, 14-15

E

ECMAScript, 24, 29, 40
ECMAScript Class, 126
ECMAScript language, 22
ECMAScript language version, 24
ECMAScript standards, 112

eight fundamental types, 40, 140
element type, 55, 72, 81
element types, 55, 80-82, 84-85
elements, 55, 80
empty array, 100
empty constructor, 134
empty object literal type, 111
empty object type `{}`, 111
empty parameter list, 93, 97
Enum, 57
`enum`, 57
enum construct, 57
`enum` declaration, 57
Enum Types, 57
`enum` values, 57
enumeration, 57
equal sign `=`, 59
equality, 42
Equality `==`, 42
equality `==`, 43
errors, 18
ES class, 53
ES module format, 29
ES module standard, 35
ES module system, 29-30, 35
ES Modules, 29
ES modules, 25-26, 29
ES2015, 29, 45, 126
ES2015 `class`, 126
ES2017, 97
ES2020, 33, 112
`es2020`, 44, 111
exception, 36, 95
`exclude`, 23
`exclude` option, 23
explicit type annotation, 61, 65-66, 79
explicit type annotations, 61
`export`, 29
`export` declaration, 30
`export` statement, 31
exported object, 32
exporting, 29
exports, 30
expression, 92, 114-115
expressions, 91
extend, 125
extending, 124
Extending Interfaces, 124
`extends` keyword, 74
`extends` syntax, 134
extensions, 22

F

`false`, 44
field, 120
Fields, 128
fields, 112
file names, 23
`files`, 23
first argument, 108
first parameter, 106
Fixed-size tuples, 80
flow analysis, 140
`for - in`, 112
`for - of`, 112
`for - of` statement, 79
`for` loop, 112
`for of` statement, 78

formal parameter, 97
formal parameter list, 91, 97
formal parameters, 91, 93, 96
`Function`, 95-96, 109
function, 17, 54, 62, 91, 95, 97, 99-100, 103, 105, 136, 145-146
Function Annotations, 61
function annotations, 53
function body, 91, 93
function body block, 92
function call, 82, 135
`Function` constructor, 95
Function context, 105
function context, 105
function declaration, 91, 93, 106
function declarations, 62
Function Definitions, 91
function expression, 92
function implementation, 101
function name, 72, 91
`Function` object, 42, 53, 95
Function Overloading, 106
Function overloading, 121
function overloading, 108, 131
function parameter, 93
function parameter list, 71
function parameter lists, 82
function parameters, 16, 62, 96, 102-103
function properties, 111
function return type, 51, 95
Function return types, 95
function return types, 120
function signature, 99, 107-108
function signatures, 107
function syntax, 120-121
`Function` type, 95, 141
function type, 53, 94-95
function type annotation, 61
Function Types, 53
Function types, 62
function types, 33, 51, 121
`Function.prototype`, 53
functional style, 146
Functions, 53, 91
functions, 16, 61-62, 72, 91

G

generated Javascript code, 64
generated Javascript program, 13-14
generated JS files, 22, 26
generic, 71, 82
generic, 73
generic anonymous function, 72
generic argument types, 72
generic array type, 72
Generic `Array<T>`, 76
generic arrow function expression, 72
generic class, 70, 139
generic class example, 73
Generic Classes, 138
Generic classes, 73
generic function, 71-72, 74
Generic function alias, 95
generic function declaration, 72
Generic Functions, 71
Generic functions, 95

generic functions, 73
generic implementation, 70
generic interface, 73
generic interfaces, 139
Generic `ReadonlyArray<T>`, 79
Generic tuples, 82
generic type, 55, 68, 74, 82, 139
generic type alias, 151
generic type argument, 77
generic type constraint, 150
Generic Type Constraints, 74
generic type parameter, 74, 152
generic type parameters, 70, 72, 95
generic type syntax, 79
generic type value, 138
Generic Types, 73
generic *vararg* function, 74
generically, 138, 152
Generics, 68
generics, 68, 70-71, 147
`get`, 117, 128
`get` syntax, 128
Getter, 117
getter, 118-119
Getter and setter, 128
getter and setter, 118-119
getter property syntax, 138
getter syntax, 118
getter type, 118
getter-setter pair, 118
Getters and Setters, 117
glob patterns, 23
Global context, 104
global context, 105
global object, 95
global scope, 29
global `this`, 104-105
global `this` object, 104
global variable, 29
`globalThis` object, 104

H

hash `#`, 127
heterogeneous arrays, 76
hierarchical type system, 121
hierarchical type systems, 122
higher-order function, 63
higher-order `map` function, 74
homogeneous, 76
HTML file, 18

I

IDE, 37
identifiers, 70
IDEs, 16
`if` branch, 142
implementation, 107-108, 146
implementation signature, 107-108
implementation signatures, 108
implementations, 134, 137
Implementing Interfaces, 135
`implements` declaration, 137
`implements` requirement, 139
`implements` syntax, 135
`import`, 29
`import - from` Declaration, 34
`import - from` declarations, 34
`import - from` statement, 34

`import` declaration, 33-34
`import` declarations, 34
`import` statements, 33
imported module, 34-35
importing, 29
importing module, 34
importing modules, 32
`in` operator, 142-143
`in` Operator Narrowing, 142
`include`, 23
`include` list, 23
index, 115
index notation, 81, 114
index signature, 116-117, 149-150
index signature syntax, 115
Index Signatures, 114
index signatures, 116-117
indexed access syntax, 150
indexed access type, 150
indexed access type syntax, 152
Indexed Access Types, 150
inequality, 42
inequality `!=`, 42-43
infer types, 140
inferred type, 38, 122, 136
infinite loop, 95
Inheritance, 130
inheritance, 124
inheritance relationships, 122
inheritance-like syntax, 124
initial name, 123
initial value, 16, 18, 36-38, 50, 60-61, 76, 81, 99, 134
initialization, 122
initializer, 97, 128
inner block, 59
input type, 152
instance, 133
instance field, 128-129, 133
Instance fields, 127
instance fields, 132
Instance getters and setters, 127
instance method, 129
Instance methods, 127
`instanceof` operator, 41, 55, 141
`instanceof` Type Guard, 141
`instanceof` type guard, 141-142
`instanceof` type guards, 145
instances, 127, 130
integer numbers, 44
`interface`, 73, 124
interface, 123-125, 135
interface declaration, 123
interface declarations, 124
interface name, 123
interface type, 104
interface type declarations, 124
Interface Types, 123
interface types, 59, 124
Interfaces, 53, 109, 123
interfaces, 73, 130, 137
interface's properties, 137
Intersection, 125
intersection, 89
intersection type, 89
intersection type, 89-90
Intersection Types, 89
intersection types, 125

item types, 76
iterable objects, 78
iterating array, 78

J

Javascript, 13-15, 17-19, 22, 25-26, 36-41, 45, 57, 103, 106, 109, 114, 117, 131, 140, 142
Javascript arrays, 55-56
Javascript classes, 126
Javascript code, 13, 19, 24, 26, 57, 146-147
Javascript community, 29
Javascript developers, 13, 26, 104
Javascript frameworks, 19
Javascript function, 95, 97
Javascript functions, 62, 97, 102
Javascript keywords, 29
Javascript libraries, 19
Javascript objects, 111, 123, 150
JavaScript program, 15
Javascript program, 13-15
Javascript programs, 13, 16
Javascript reference, 38, 99
Javascript runtime, 13
Javascript runtimes, 29, 131
Javascript semantics, 106
Javascript statement, 54
Javascript Types, 40
JS class, 134
JS code, 26
JS files, 25
JS project, 26
JS-private, 132
JSON Configuration File, 22

K

key type, 150
key types, 116
`keyof`, 147
`keyof` operator, 149
`keyof` Type Operator, 149
keys, 117
keyword `abstract`, 133
keyword `class`, 126
keyword `const`, 65
keyword `extends`, 74
keyword `function`, 91
keyword `get`, 118
keyword `interface`, 123
keyword `set`, 118
keyword `type`, 59

L

labeled tuple, 81, 84
labeled tuple types, 82
Lambda function, 92
last element, 84
last parameter, 99
left-to-right associative, 152
legacy code, 26
`length` property, 85
`let`, 36, 39, 60
`let` Declaration, 37
`let` declaration, 37-38
`let` declarations, 31
`let` variable, 37-38
`let` variable declaration, 38

lexical scope, 104
libraries, 19
literal syntax, 44, 77, 79
literal type, 46, 60-61, 65, 119, 148
literal type expressions, 65
Literal Types, 45
literal types, 46, 58, 65
literal value, 46
Literal values, 45
local variable declarations, 18
loop variable, 78

M

`map` function, 63
mapped type, 152
mapped type syntax, 152
Mapped Types, 152
Mapped types, 152
mapping-like syntax, 152
matching setter, 118
member access syntax, 114
Member Methods, 120
member type, 86-88, 90
member types, 89-90, 123, 142, 144-145
Member visibility, 131
members, 125
members types, 89
method, 146
method members, 121
method or field, 127
Method overloading, 121
Method properties, 120
method properties, 117, 120
method property, 106
Methods, 121, 129
middle element, 84
minification, 19
modern ECMAScript, 26
modern Javascript, 102, 105, 112
modern Javascript projects, 19
`module`, 25
module, 29-30
module configuration value, 29
Module Exports, 30
module format, 25, 29, 35
module formats, 29
Module `import`, 34
Module Imports, 33
module output format, 29
module resolution rules, 30
module system, 29
module's default, 32
module's `default`, 33
module's default export, 32-33
multiple interfaces, 137
multiple variables, 39

N

Name list `export`, 32
Name list `import`, 35
named argument, 103
named class declaration, 127
named expression, 126
named function, 32
named properties, 116
named property, 117
named types, 58

`namespace`, 35
namespace alias, 35
Namespace `import`, 35
namespaces, 35
narrowest type, 51
Narrowing, 87-88
narrowing, 48, 87, 89
nested conditional type, 152
`never`, 51-52, 89, 95
`never` Type, 51
`never` type, 51
new `class`, 126
`new` constructor call, 131
`new` operator, 42
new type, 33, 41, 80, 147
new types, 123
Node.js, 20
Node.js REPL, 105
`noImplicitAny`, 27-28, 96
`noImplicitThis`, 27-28, 105
non-default exports, 30
non-iterable object, 79
non-null assertion, 67, 108
Non-Null Assertions, 67
non-null assertions, 67
non-strict mode, 36, 104-105
non-trailing elements, 84
NPM, 25
npm, 20
NPM package, 29
NPM package repository, 29
npm project, 21
`null`, 41, 49-50
`null` and `undefined`, 40, 49-50, 67
`null` or `undefined`, 27, 50, 67, 99-100
`null` type, 49
null-related errors, 27
`number`, 18, 44-45, 58-59
`number` array, 63
`Number` for `number`, 40
`number` type, 16, 44, 59
`number` value, 18

O

obfuscation, 19
`Object`, 52, 96
Object, 109
`object`, 52, 109
object, 53, 105, 110, 136
object, 109
`Object` constructor, 109
object instance, 129
object literal, 109, 122
object literal type, 32, 58-59, 104, 110, 121, 123-124
object literal type syntax, 109
object literal type `{}`, 111
Object Literal Types, 109
object literal types, 59, 65, 90, 109, 123-125, 130, 135, 145
object literal value, 66
object literal values, 137
Object literals, 65
object literals, 110, 128
object structures, 112
`object` Type, 52
`object` type, 52-53, 109, 111
object type, 66, 109-110, 115-116, 123-

124, 149-150
Object Type Members, 111
Object Types, 32, 53
object types, 53, 109, 115, 123
object-based, 40
`Object.prototype`, 53, 106
`Object.toString` method, 138
Objects, 109
objects, 52
objects, 102, 109, 126
OOP programming languages, 135
operation, 89
operations, 14, 86
optional, 82, 84, 108, 111-112, 120, 131-132
optional chaining, 112
optional element, 84
optional function parameters, 111-112
optional method property, 120
optional parameter, 98-99
Optional Parameters, 98
optional parameters, 97, 108
optional parameters with `?`, 99
Optional properties, 111
optional properties, 112
optional tuple elements, 112
options, 22
outer scope, 59
output module format, 25
overload a function, 107
overload signature, 107-108
Overload signatures, 107
overload signatures, 107-108
overloaded, 109
overloaded function, 108

P

parameter, 62, 97, 103
Parameter Destructuring, 102
Parameter initializers, 97
Parameter List, 96
parameter list, 97
parameter lists, 107
Parameter properties, 132
parameter property syntax, 139
parameter type, 63
parameterized type, 76
parameterized types, 68
parameters, 16, 61-62, 93-94
Parameters with initializers, 97
parent interface, 125
parentheses `()`, 62, 91, 94
plain Javascript code, 47
polyfills, 24
predefined objects, 40
predefined type, 109
primitive type, 58, 109
Primitive Types, 44
primitive types, 40, 44, 52, 58, 111, 121
private, 127
private, 131
`private`, 131
`private` field, 132
private field, 132, 138
private instance field, 128
private members, 131

private `static` field, 128
product types, 86
programming language, 13
programming languages, 13, 108
project setup, 18
Properties, 113
properties, 109, 114-116, 127, 133
properties of interfaces, 135
property, 117-118
property access syntax, 117
property key, 115
property keys, 152
property members, 111
property name, 111
property names, 109, 122
property separators, 110
property type, 117
property value, 117
`protected`, 131
`protected` field, 132
protected properties, 134
protected property, 132
prototype, 41
prototype chain, 41
prototype chain, 52-53, 109, 141
prototype property, 41
prototypes, 126
pseudo properties, 117
pseudo-property, 118, 138
public, 127
public, 131, 134-135
`public`, 131
public by default, 132
public field, 69, 135
`public` instance field, 132
public instance fields, 132
public members, 131
public method, 69
`public` modifier, 132
public property, 132
publicly accessible, 133
Pure type declarations, 130
`push` method, 85
Python, 18

Q

question mark `?`, 112

R

rapid iteration, 18
React, 19, 65
readonly, 85
`readonly`, 113, 152
readonly array types, 79
Readonly arrays, 79
`readonly` keyword, 85
`readonly` modifier, 152
Readonly properties, 113
`readonly` properties, 65, 113
`readonly` property, 66, 118
readonly property, 113, 118
`readonly T[]`, 79
`readonly` tuple, 65
readonly tuple, 85
readonly tuple, 85
`readonly` tuple type, 66
Readonly Tuples, 85
Readonly tuples, 85

readonly tuples, 56, 85
`readonly` tuples, 65
`ReadonlyArray`, 79, 85
`ReadonlyArray` type, 79
`ReadonlyArray<T>`, 79
REPL, 105
required fields, 112
required property, 117
rest element, 83
rest element types, 85
Rest Parameter, 99
rest parameter, 74, 96-97, 99, 101
rest parameter syntax, 99
rest parameters, 101
resulting type, 152
`return` statement, 50, 95
return type, 51, 69-70, 89, 95, 131
return value, 16, 50, 62-63, 95
return values, 16, 61
roperty syntax, 117
run time, 15, 41, 47, 106, 142
run time error, 47, 103, 141
run-time errors, 132
run-time exception, 79
run-time type, 140
runtime, 143
runtime assertion, 64
runtime behavior, 64, 142
runtime environments, 104
runtime error, 67
runtime errors, 15
runtime support, 146
runtime-private, 131

S

same types, 122
scope, 59
scoping, 58
scoping rules, 38
script, 29
script tag, 18
sequence of items, 54, 76
`set`, 117, 128
`set` syntax, 128
setter, 118
setter parameter, 118
setter syntax, 118
setter type, 118-119
setters, 117
shadowing, 58
shape, 123
shapes, 121
simple TS code, 20
single parameter, 94
singleton types, 49
`slice` method, 78
source code repository, 25
source files, 23
source JS files, 26
specified type, 16, 38
spread element types, 101
spread operator, 84
spread operator `...`, 83
square brackets `[]`, 80
statement, 91, 93, 138-139
statements, 91, 93, 129
`static`, 127
static analysis, 18

`static` block, 127
Static blocks, 129
`static` field, 128
`static` fields, 127
static flow analysis, 140, 142
`static` getters and setters, 127
static initialization block, 129
`static` initialization blocks, 127
`static` keyword, 127
`static` methods, 127
Static properties, 127
static tooling, 15
static type analyzer, 37
static type checker, 48, 67, 69, 132
Static type checking, 16
static type checking, 15, 26, 46, 48, 100, 137
static type error, 149
static type information, 15, 76
static types, 43
Static Typing, 16
static typing, 13, 15
statically annotated, 64
statically type checked, 47
statically typed, 13, 15
`strict`, 27
Strict Equality, 42
strict equality `===`, 42
strict inequality `!==`, 42
strict mode, 22, 105
strict mode code, 126
strict mode variant, 26
`strict` options, 27
strict type checking, 27, 43
`strictBindCallApply`, 27
`strictFunctionTypes`, 27
strictness settings, 38
`strictNullChecks`, 27, 38, 50
`strictPropertyInitialization`, 27
`string`, 44-45, 59, 115
`String` for `string`, 40
string literal type, 147
string literal types, 147
`string` type, 44
strongly typed, 121
strongly typed language, 69
strongly typed languages, 17
structural subtype, 124-125
Structural Subtyping, 121
structural typing, 136
structurally compatible, 136
structurally narrower, 137
structurally typed, 142
structure, 110, 152
structure-based type system, 123
structures, 121
subclass, 134
subclasses, 132
subtype, 121-122
sum types, 85
`super()`, 134
supertype, 109, 121
`switch` branch, 89
`Symbol`, 45
`symbol`, 45
`Symbol` for `symbol`, 40
`Symbol` type, 45

`symbol` type, 45
syntactic shortcut, 124

T

`target`, 24
`target` option, 24
target type, 59
template literal type, 147
template literal type syntax, 147
Template Literal Types, 147
template string syntax, 147
ternary operator, 152
`this`, 28, 104-106
`this` binding, 104
`this` function parameter, 28
`this` object, 105
`this` operator, 104
`this` Parameter, 104
`this` parameter, 106, 131
`this` value, 104
top-level base type, 121
Top-level options, 23
top-level variables, 38
`toString` method, 106
trailing comma, 97
trailing elements, 84
trailing optional, 85
trailing portion, 97
trailing separator, 110
transpilation, 15, 56
transpiled output, 24
`true`, 44
TS and JS source files, 25-26
TS class, 134
TS compiler, 18
TS files, 22
TS project, 26
TS-private, 132
TS-protected, 132
TS-specific, 133
tsc, 20-22, 26-27
`tsc`, 22
tsc --all, 22
tsc --help, 22
tsc --init, 21
`tsc --init`, 23
tsc command, 20, 22
tsconfig.json, 21
tsconfig.json file, 21-23, 29
tuple, 56, 80, 84
tuple destructuring, 119
tuple element, 81
tuple elements, 82
tuple literal, 80
tuple type, 56, 80-81, 83-85, 101
Tuple Types, 55
Tuple types, 82
tuple types, 56, 82, 86, 101-102
`tuple` types, 80
Tuples, 56, 80
tuples, 55-56
two-element tuple, 81
type, 14-15
`type`, 58
type alias, 58-59, 81, 94, 104, 124, 139, 149
Type alias declarations, 58
type alias declarations, 59

Type aliases, 58-59, 86
type aliases, 58-59
type aliasing, 123
Type Annotation, 37
type annotation, 31, 38-39, 46, 50, 60-61, 69, 74, 76, 101
Type annotations, 16
type annotations, 16, 30, 62, 64, 66, 79, 106
type argument, 73
type arguments, 68, 74, 139
type assertion, 30, 33, 48-49, 64-65, 67, 78, 101, 143
Type Assertions, 64
type assertions, 64-65, 67
Type checking, 26
type checking, 49
type checking error, 81
type checking errors, 46
type compatibility, 47
type constraint, 74, 138
type constraints, 74
type context, 148
type declarations, 19
type definition file, 20
type definition files, 19
type erasure, 15
type error, 150
type expressions, 147
type guard, 98, 108, 119, 141, 148
type guards, 140
Type inference, 18, 70
type inference, 31, 39, 79
type information, 15, 28
type inheritance, 74
type literals, 58
type name, 65
type names, 136
Type narrowing, 143
type narrowing, 87, 142-143
type narrowing, 140
type `Object`, 109
type parameter, 70, 73-74, 138
type parameter names, 70
type parameters, 68, 71-74
type safety, 69
type system, 121
type systems, 121
type-checking options, 27
`typeof`, 147
`typeof` expressions, 140
`typeof` operator, 41, 55, 119, 140, 148
`typeof` Type Guard, 140
`typeof` type guard, 87, 140-141
`typeof` Type Operator, 148
`typeof(null)`, 41
types, 14-15, 109
TypeScript, 140
Typescript, 13, 15-22, 24-26, 28, 36-37, 39, 45-46, 103, 105-106, 108-109, 112, 118, 122-124, 130-131, 140, 142
Typescript Class, 53, 130
Typescript `class`, 126, 135, 138
Typescript classes, 69
Typescript code, 57, 105
Typescript Compiler, 20
Typescript compiler, 15, 17, 20, 60, 62,

132
Typescript core libraries, 20
Typescript expressions, 20
Typescript file, 29
Typescript files, 21
Typescript identifier, 34
Typescript language, 20
Typescript Modules, 29
Typescript Namespaces, 35
Typescript namespaces, 35
Typescript object types, 111
Typescript package, 20
typescript package, 20
TypeScript Playground, 20
Typescript program, 13-15
Typescript programming, 147
Typescript programs, 13, 16
Typescript project, 21
Typescript project root, 23
Typescript REPL, 20
Typescript source files, 22-23
Typescript statement, 54
Typescript statements, 20
Typescript transpilation, 19
Typescript tuple types, 82
Typescript's enum, 57

U

`undefined`, 37-38, 49-50, 52, 89, 95, 105, 112
`undefined` return type, 95
`undefined` type, 49-50, 84, 98
union, 87, 143
union return type, 95
union type, 50, 55, 58, 86-88, 142, 148-149
union type, 85
union type parameters, 108
Union Types, 85
Union types, 85
union types, 45, 86, 147
unions, 88
universal type, 95
`unknown`, 47-48, 52, 55, 64, 76, 143
`unknown` Type, 47
`unknown` type, 43, 47-49
`unknown` value, 48
`unknown[]`, 55
untyped function call, 96
`useUnknownInCatchVariable`, 27

V

valid value, 149
valid values, 83
validation logic, 117
value equality, 119
`var`, 36, 39, 60
`var` Declaration, 38
`var` declaration, 38
variable, 14, 16-18, 28, 36-39, 60-62, 64, 79
Variable Annotations, 60
variable declaration, 60
Variable `export`, 31
variable type annotation, 61
variable type annotations, 62
Variables, 36
variables, 14-16, 37, 39

variables and values, 14
variadic function, 100
variadic functions, 82, 99
Variadic tuples, 82
variadic tuples, 82
variadic tuples, 83
variant, 88
variant of Typescript, 22
variants, 22, 88
variants, 22, 88-89
version control system, 25
visibility modifiers, 131
`void`, 52, 95, 120
`void` return type, 69
`void` Type, 51
`void` type, 51
Vuejs, 19

W

watch mode, 22
weakly typed languages, 68
Web browsers, 29
What is Typescript?, 13
Why Generics?, 68
wildcard `*` syntax, 35
wildcard characters, 23
`Window` object, 104
write access, 114

Y

yarn, 20

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

About the Series

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages*. We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, Javascript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

- Go Mini Reference [https://www.amazon.com/dp/B09V5QXTCC/]
- Modern C# Mini Reference [https://www.amazon.com/dp/B0B57PXLFC/]
- Python Mini Reference [https://www.amazon.com/dp/B0B2QJD6P8/]
- Typescript Mini Reference [https://www.amazon.com/dp/B0B54537JK/]
- Rust Mini Reference [https://www.amazon.com/dp/B09Y74PH2B/]
- C++20 Mini Reference [https://www.amazon.com/dp/B0B5YLXLB3/]
- Modern Java Mini Reference [https://www.amazon.com/dp/B0B75PCHW2/]
- Julia Mini Reference [https://www.amazon.com/dp/B0B6PZ2BCJ/]
- Javascript Mini Reference [https://www.amazon.com/dp/B0B75RZLRB/]
- Haskell Mini Reference [https://www.amazon.com/dp/B09X8PLG9P/]
- Scala 3 Mini Reference [https://www.amazon.com/dp/B0B95Y6584/]
- Lua Mini Reference [https://www.amazon.com/dp/B09V95T452/]

Community Support

We are building a website for programmers, from beginners to more experienced. It covers various coding-related topics from algorithms to machine learning, and from design patterns to cybersecurity, and more. You can also find some sample code in the GitLab repositories.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Please join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books Press**, including free, or discounted, book promotions. If we find any significant errors in the book, then we will send you an updated version of the book (in PDF). Advance review copies will be made available to select members on the list before new books are published.

Request for Feedback

If you find any errors or typos, or if any part of the book is not very clear to you, or if you have any general suggestions or comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to address the issues that are brought to our attention.

- feedback@codingbookspress.com

Please note that creating and publishing quality books takes a great deal of time and effort, and we really appreciate the readers' feedback.

Revision 1.0.8, 2023-05-14