

TypeScript for Functional Programmers

- [Prerequisites](#)
- [Concepts not in Haskell](#)
- [Built-in types](#)
- [Other important TypeScript types](#)
- [Boxed types](#)
- [Gradual typing](#)
- [Structural typing](#)
- [Unions](#)
- [Intersections](#)
- [Unit types](#)
- [Concepts similar to Haskell](#)
- [Contextual typing](#)
- [Type aliases](#)
- [Discriminated Unions](#)
- [Type Parameters](#)
- [Higher-kinded types](#)
- [Point-free programming](#)
- [Module system](#)
- [readonly and const](#)
- [Next Steps](#)

TypeScript began its life as an attempt to bring traditional object-oriented types to JavaScript so that the programmers at Microsoft could bring traditional object-oriented programs to the web. As it has developed, TypeScript’s type system has evolved to model code written by native JavaScripters. The resulting system is powerful, interesting and messy.

This introduction is designed for working Haskell or ML programmers who want to learn TypeScript. It describes how the type system of TypeScript differs from Haskell’s type system. It also describes unique features of TypeScript’s type system that arise from its modelling of JavaScript code.

This introduction does not cover object-oriented programming. In practice, object-oriented programs in TypeScript are similar to those in other popular languages with OO features.

Prerequisites

In this introduction, I assume you know the following:

- How to program in JavaScript, the good parts.
- Type syntax of a C-descended language.

If you need to learn the good parts of JavaScript, read [JavaScript: The Good Parts](#). You may be able to skip the book if you know how to write programs in a call-by-value lexically scoped language with lots of mutability and not much else. [R⁴RS Scheme](#) is a good example.

[The C++ Programming Language](#) is a good place to learn about C-style type syntax. Unlike C++, TypeScript uses postfix types, like so: `x: string` instead of `string x`.

Concepts not in Haskell

Built-in types

JavaScript defines 8 built-in types:

Type	Explanation
Number	a double-precision IEEE 754 floating point.
String	an immutable UTF-16 string.
BigInt	integers in the arbitrary precision format.
Boolean	true and false.

Symbol	a unique value usually used as a key.
Null	equivalent to the unit type.
Undefined	also equivalent to the unit type.
Object	similar to records.

[See the MDN page for more detail.](#)

TypeScript has corresponding primitive types for the built-in types:

- number
- string
- bigint
- boolean
- symbol
- null
- undefined
- object

Other important TypeScript types

Type	Explanation
unknown	the top type.
never	the bottom type.
object literal	eg { property: Type }
void	a subtype of undefined intended for use as a return type.
T[]	mutable arrays, also written Array<T>
[T, T]	tuples, which are fixed-length but mutable
(t: T) => U	functions

Notes:

- Function syntax includes parameter names. This is pretty hard to get used to!

```
let fst: (a: any, b: any) => any = (a, b) => a;
// or more precisely:
let fst: <T, U>(a: T, b: U) => T = (a, b) => a;
```

- Object literal type syntax closely mirrors object literal value syntax:

```
let o: { n: number; xs: object[] } = { n: 1, xs: [] };
```

- [T, T] is a subtype of T[] . This is different than Haskell, where tuples are not related to lists.

Boxed types

JavaScript has boxed equivalents of primitive types that contain the methods that programmers associate with those types. TypeScript reflects this with, for example, the difference between the primitive type number and the boxed type Number . The boxed types are rarely needed, since their methods return primitives.

```
(1).toExponential();
// equivalent to
Number.prototype.toExponential.call(1);
```

Note that calling a method on a numeric literal requires it to be in parentheses to aid the parser.

Gradual typing

TypeScript uses the type `any` whenever it can't tell what the type of an expression should be. Compared to `Dynamic`, calling `any` a type is an overstatement. It just turns off the type checker wherever it appears. For example, you can push any value into an `any[]` without marking the value in any way:

```
// with "noImplicitAny": false in tsconfig.json, anys: any[]
const anys = [];
anys.push(1);
anys.push("oh no");
anys.push({ anything: "goes" });
```

And you can use an expression of type `any` anywhere:

```
anys.map(anys[1]); // oh no, "oh no" is not a function
```

`any` is contagious, too — if you initialise a variable with an expression of type `any`, the variable has type `any` too.

```
let sepsis = anys[0] + anys[1]; // this could mean anything
```

To get an error when TypeScript produces an `any`, use `"noImplicitAny": true`, or `"strict": true` in `tsconfig.json`.

Structural typing

Structural typing is a familiar concept to most functional programmers, although Haskell and most MLs are not structurally typed. Its basic form is pretty simple:

```
// @strict: false
let o = { x: "hi", extra: 1 }; // ok
let o2: { x: string } = o; // ok
```

Here, the object literal `{ x: "hi", extra: 1 }` has a matching literal type `{ x: string, extra: number }`. That type is assignable to `{ x: string }` since it has all the required properties and those properties have assignable types. The extra property doesn't prevent assignment, it just makes it a subtype of `{ x: string }`.

Named types just give a name to a type; for assignability purposes there's no difference between the type alias `One` and the interface type `Two` below. They both have a property `p: string`. (Type aliases behave differently from interfaces with respect to recursive definitions and type parameters, however.)

```
type One = { p: string };
interface Two {
  p: string;
}
class Three {
  p = "Hello";
}

let x: One = { p: "hi" };
let two: Two = x;
two = new Three();
```

Unions

In TypeScript, union types are untagged. In other words, they are not discriminated unions like `data` in Haskell. However, you can often discriminate types in a union using built-in tags or other properties.

```
function start(
  arg: string | string[] | ((() => string) | { s: string })
): string {
  // this is super common in JavaScript
  if (typeof arg === "string") {
    return commonCase(arg);
  } else if (Array.isArray(arg)) {
    return arg.map(commonCase).join(",");
  } else if (typeof arg === "function") {
    return commonCase(arg());
  } else {
    return commonCase(arg.s);
  }
}
```

```
    }

    function commonCase(s: string): string {
      // finally, just convert a string to another string
      return s;
    }
  }
}
```

`string`, `Array` and `Function` have built-in type predicates, conveniently leaving the object type for the `else` branch. It is possible, however, to generate unions that are difficult to differentiate at runtime. For new code, it's best to build only discriminated unions.

The following types have built-in predicates:

Type	Predicate
<code>string</code>	<code>typeof s === "string"</code>
<code>number</code>	<code>typeof n === "number"</code>
<code>bigint</code>	<code>typeof m === "bigint"</code>
<code>boolean</code>	<code>typeof b === "boolean"</code>
<code>symbol</code>	<code>typeof g === "symbol"</code>
<code>undefined</code>	<code>typeof undefined === "undefined"</code>
<code>function</code>	<code>typeof f === "function"</code>
<code>array</code>	<code>Array.isArray(a)</code>
<code>object</code>	<code>typeof o === "object"</code>

Note that functions and arrays are objects at runtime, but have their own predicates.

Intersections

In addition to unions, TypeScript also has intersections:

```
type Combined = { a: number } & { b: string };
type Conflicting = { a: number } & { a: string };
```

`Combined` has two properties, `a` and `b`, just as if they had been written as one object literal type. Intersection and union are recursive in case of conflicts, so `Conflicting.a: number & string`.

Unit types

Unit types are subtypes of primitive types that contain exactly one primitive value. For example, the string `"foo"` has the type `"foo"`. Since JavaScript has no built-in enums, it is common to use a set of well-known strings instead. Unions of string literal types allow TypeScript to type this pattern:

```
declare function pad(s: string, n: number, direction: "left" | "right"): string;
pad("hi", 10, "left");
```

When needed, the compiler *widens* — converts to a supertype — the unit type to the primitive type, such as `"foo"` to `string`. This happens when using mutability, which can hamper some uses of mutable variables:

```
let s = "right";
pad("hi", 10, s); // error: 'string' is not assignable to '"left" | "right"'
```

Argument of type 'string' is not assignable to parameter of type '"left" | "right"'.

Here's how the error happens:

- `"right": "right"`
- `s: string` because `"right"` widens to `string` on assignment to a mutable variable.
- `string` is not assignable to `"left" | "right"`

You can work around this with a type annotation for `s`, but that in turn prevents assignments to `s` of variables that are not of type `"left" | "right"`.

```
let s: "left" | "right" = "right";
pad("hi", 10, s);
```

Concepts similar to Haskell

Contextual typing

TypeScript has some obvious places where it can infer types, like variable declarations:

```
let s = "I'm a string!";
```

But it also infers types in a few other places that you may not expect if you've worked with other C-syntax languages:

```
declare function map<T, U>(f: (t: T) => U, ts: T[]): U[];
let sns = map((n) => n.toString(), [1, 2, 3]);
```

Here, `n: number` in this example also, despite the fact that `T` and `U` have not been inferred before the call. In fact, after `[1, 2, 3]` has been used to infer `T=number`, the return type of `n => n.toString()` is used to infer `U=string`, causing `sns` to have the type `string[]`.

Note that inference will work in any order, but intellisense will only work left-to-right, so TypeScript prefers to declare `map` with the array first:

```
declare function map<T, U>(ts: T[], f: (t: T) => U): U[];
```

Contextual typing also works recursively through object literals, and on unit types that would otherwise be inferred as `string` or `number`. And it can infer return types from context:

```
declare function run<T>(thunk: (t: T) => void): T;
let i: { inference: string } = run((o) => {
  o.inference = "INSERT STATE HERE";
});
```

The type of `o` is determined to be `{ inference: string }` because

1. Declaration initialisers are contextually typed by the declaration's type: `{ inference: string }`.
2. The return type of a call uses the contextual type for inferences, so the compiler infers that `T={ inference: string }`.
3. Arrow functions use the contextual type to type their parameters, so the compiler gives `o: { inference: string }`.

And it does so while you are typing, so that after typing `o.`, you get completions for the property `inference`, along with any other properties you'd have in a real program. Altogether, this feature can make TypeScript's inference look a bit like a unifying type inference engine, but it is not.

Type aliases

Type aliases are mere aliases, just like `type` in Haskell. The compiler will attempt to use the alias name wherever it was used in the source code, but does not always succeed.

```
type Size = [number, number];
let x: Size = [101.1, 999.9];
```

The closest equivalent to `newtype` is a *tagged intersection*:

```
type FString = string & { __compileTimeOnly: any };
```

An `FString` is just like a normal string, except that the compiler thinks it has a property named `__compileTimeOnly` that doesn't actually exist. This means that `FString` can still be assigned to `string`, but not the other way round.

Discriminated Unions

The closest equivalent to `data` is a union of types with discriminant properties, normally called discriminated unions in TypeScript:

```
type Shape =  
  | { kind: "circle"; radius: number }  
  | { kind: "square"; x: number }  
  | { kind: "triangle"; x: number; y: number };
```

Unlike Haskell, the tag, or discriminant, is just a property in each object type. Each variant has an identical property with a different unit type. This is still a normal union type; the leading `|` is an optional part of the union type syntax. You can discriminate the members of the union using normal JavaScript code:

```
type Shape =  
  | { kind: "circle"; radius: number }  
  | { kind: "square"; x: number }  
  | { kind: "triangle"; x: number; y: number };  
  
function area(s: Shape) {  
  if (s.kind === "circle") {  
    return Math.PI * s.radius * s.radius;  
  } else if (s.kind === "square") {  
    return s.x * s.x;  
  } else {  
    return (s.x * s.y) / 2;  
  }  
}
```

Note that the return type of `area` is inferred to be `number` because TypeScript knows the function is total. If some variant is not covered, the return type of `area` will be `number | undefined` instead.

Also, unlike Haskell, common properties show up in any union, so you can usefully discriminate multiple members of the union:

```
function height(s: Shape) {  
  if (s.kind === "circle") {  
    return 2 * s.radius;  
  } else {  
    // s.kind: "square" | "triangle"  
    return s.x;  
  }  
}
```

Type Parameters

Like most C-descended languages, TypeScript requires declaration of type parameters:

```
function liftArray<T>(t: T): Array<T> {  
  return [t];  
}
```

There is no case requirement, but type parameters are conventionally single uppercase letters. Type parameters can also be constrained to a type, which behaves a bit like type class constraints:

```
function firstish<T extends { length: number }>(t1: T, t2: T): T {  
  return t1.length > t2.length ? t1 : t2;  
}
```

TypeScript can usually infer type arguments from a call based on the type of the arguments, so type arguments are usually not needed.

Because TypeScript is structural, it doesn't need type parameters as much as nominal systems. Specifically, they are not needed to make a function polymorphic. Type parameters should only be used to *propagate* type information, such as constraining parameters to be the same type:

```
function length<T extends ArrayLike<unknown>>(t: T): number {}  
function length(t: ArrayLike<unknown>): number {}
```

In the first `length`, `T` is not necessary; notice that it's only referenced once, so it's not being used to constrain the type of the return value or other parameters.

Higher-kinded types

TypeScript does not have higher kinded types, so the following is not legal:

```
function length<T extends ArrayLike<unknown>, U>(m: T<U>) {}
```

Point-free programming

Point-free programming — heavy use of currying and function composition — is possible in JavaScript, but can be verbose. In TypeScript, type inference often fails for point-free programs, so you'll end up specifying type parameters instead of value parameters. The result is so verbose that it's usually better to avoid point-free programming.

Module system

JavaScript's modern module syntax is a bit like Haskell's, except that any file with `import` or `export` is implicitly a module:

```
import { value, Type } from "npm-package";
import { other, Types } from "../local-package";
import * as prefix from "../lib/third-package";
```

You can also import commonjs modules — modules written using nodejs' module system:

```
import f = require("single-function-package");
```

You can export with an export list:

```
export { f };
function f() {
  return g();
}
function g() {} // g is not exported
```

Or by marking each export individually:

```
export function f { return g() }
function g() { }
```

The latter style is more common but both are allowed, even in the same file.

`readonly` and `const`

In JavaScript, mutability is the default, although it allows variable declarations with `const` to declare that the *reference* is immutable. The referent is still mutable:

```
const a = [1, 2, 3];
a.push(102); // ):
a[0] = 101; // D:
```

TypeScript additionally has a `readonly` modifier for properties.

```
interface Rx {
  readonly x: number;
}
let rx: Rx = { x: 1 };
rx.x = 12; // error
```

It also ships with a mapped type `Readonly<T>` that makes all properties `readonly`:

```
interface X {
  x: number;
}
let rx: Readonly<X> = { x: 1 };
rx.x = 12; // error
```

And it has a specific `ReadonlyArray<T>` type that removes side-affecting methods and prevents writing to indices of the array, as well as special syntax for this type:

```
let a: ReadonlyArray<number> = [1, 2, 3];
let b: readonly number[] = [1, 2, 3];
a.push(102); // error
b[0] = 101; // error
```

You can also use a const-assertion, which operates on arrays and object literals:

```
let a = [1, 2, 3] as const;
a.push(102); // error
a[0] = 101; // error
```

However, none of these options are the default, so they are not consistently used in TypeScript code.

Next Steps

This doc is a high level overview of the syntax and types you would use in everyday code. From here you should:

- Read the full Handbook [from start to finish](#) (30m)
- Explore the [Playground examples](#)

TypeScript for JavaScript Programmers

- [Types by Inference](#)
- [Defining Types](#)
- [Composing Types](#)
- [Unions](#)
- [Generics](#)
- [Structural Type System](#)
- [Next Steps](#)

TypeScript stands in an unusual relationship to JavaScript. TypeScript offers all of JavaScript's features, and an additional layer on top of these: TypeScript's type system.

For example, JavaScript provides language primitives like `string` and `number`, but it doesn't check that you've consistently assigned these. TypeScript does.

This means that your existing working JavaScript code is also TypeScript code. The main benefit of TypeScript is that it can highlight unexpected behavior in your code, lowering the chance of bugs.

This tutorial provides a brief overview of TypeScript, focusing on its type system.

Types by Inference

TypeScript knows the JavaScript language and will generate types for you in many cases. For example in creating a variable and assigning it to a particular value, TypeScript will use the value as its type.

```
let helloWorld = "Hello World";  
let helloWorld: string
```

By understanding how JavaScript works, TypeScript can build a type-system that accepts JavaScript code but has types. This offers a type-system without needing to add extra characters to make types explicit in your code. That's how TypeScript knows that `helloWorld` is a `string` in the above example.

You may have written JavaScript in Visual Studio Code, and had editor auto-completion. Visual Studio Code uses TypeScript under the hood to make it easier to work with JavaScript.

Defining Types

You can use a wide variety of design patterns in JavaScript. However, some design patterns make it difficult for types to be inferred automatically (for example, patterns that use dynamic programming). To cover these cases, TypeScript supports an extension of the JavaScript language, which offers places for you to tell TypeScript what the types should be.

For example, to create an object with an inferred type which includes `name: string` and `id: number`, you can write:

```
const user = {  
  name: "Hayes",  
  id: 0,  
};
```

You can explicitly describe this object's shape using an `interface` declaration:

```
interface User {  
  name: string;  
  id: number;  
}
```

You can then declare that a JavaScript object conforms to the shape of your new `interface` by using syntax like `: TypeName` after a variable declaration:

```
const user: User = {  
  name: "Hayes",  
  id: 0,  
};
```

```
};
```

If you provide an object that doesn't match the interface you have provided, TypeScript will warn you:

```
interface User {  
  name: string;  
  id: number;  
}
```

```
const user: User = {  
  username: "Hayes",
```

```
Type '{ username: string; id: number; }' is not assignable to type 'User'.  
  Object literal may only specify known properties, and 'username' does not exist in type 'User'.
```

```
  id: 0,  
};
```

Since JavaScript supports classes and object-oriented programming, so does TypeScript. You can use an interface declaration with classes:

```
interface User {  
  name: string;  
  id: number;  
}
```

```
class UserAccount {  
  name: string;  
  id: number;  
  
  constructor(name: string, id: number) {  
    this.name = name;  
    this.id = id;  
  }  
}
```

```
const user: User = new UserAccount("Murphy", 1);
```

You can use interfaces to annotate parameters and return values to functions:

```
function getAdminUser(): User {  
  //...  
}  
  
function deleteUser(user: User) {  
  // ...  
}
```

There are already a small set of primitive types available in JavaScript: `boolean`, `bigint`, `null`, `number`, `string`, `symbol`, and `undefined`, which you can use in an interface. TypeScript extends this list with a few more, such as `any` (allow anything), `unknown` (ensure someone using this type declares what the type is), `never` (it's not possible that this type could happen), and `void` (a function which returns `undefined` or has no return value).

You'll see that there are two syntaxes for building types: [Interfaces and Types](#). You should prefer `interface`. Use `type` when you need specific features.

Composing Types

With TypeScript, you can create complex types by combining simple ones. There are two popular ways to do so: with Unions, and with Generics.

Unions

With a union, you can declare that a type could be one of many types. For example, you can describe a `boolean` type as being either `true` or `false`:

```
type MyBool = true | false;
```

Note: If you hover over `MyBool` above, you'll see that it is classed as `boolean`. That's a property of the Structural Type System. More on this below.

A popular use-case for union types is to describe the set of `string` or `number` [literals](#) that a value is allowed to be:

```
type WindowStates = "open" | "closed" | "minimized";
```

```
type LockStates = "locked" | "unlocked";
type OddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Unions provide a way to handle different types too. For example, you may have a function that takes an `array` or a `string`:

```
function getLength(obj: string | string[]) {
  return obj.length;
}
```

To learn the type of a variable, use `typeof`:

Type	Predicate
string	typeof s === "string"
number	typeof n === "number"
boolean	typeof b === "boolean"
undefined	typeof undefined === "undefined"
function	typeof f === "function"
array	Array.isArray(a)

For example, you can make a function return different values depending on whether it is passed a string or an array:

```
function wrapInArray(obj: string | string[]) {
  if (typeof obj === "string") {
    return [obj];
  } else {
    return obj;
  }
}
```

(parameter) obj: string

Generics

Generics provide variables to types. A common example is an array. An array without generics could contain anything. An array with generics can describe the values that the array contains.

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;
```

You can declare your own types that use generics:

```
interface Backpack<Type> {
  add: (obj: Type) => void;
  get: () => Type;
}

// This line is a shortcut to tell TypeScript there is a
// constant called `backpack`, and to not worry about where it came from.
declare const backpack: Backpack<string>;

// object is a string, because we declared it above as the variable part of Backpack.
const object = backpack.get();

// Since the backpack variable is a string, you can't pass a number to the add function.
backpack.add(23);
```

Argument of type 'number' is not assignable to parameter of type 'string'.

Structural Type System

One of TypeScript’s core principles is that type checking focuses on the *shape* that values have. This is sometimes called “duck typing” or “structural typing”.

In a structural type system, if two objects have the same shape, they are considered to be of the same type.

```
interface Point {  
  x: number;  
  y: number;  
}  
  
function logPoint(p: Point) {  
  console.log(`${p.x}, ${p.y}`);  
}  
  
// logs "12, 26"  
const point = { x: 12, y: 26 };  
logPoint(point);
```

The `point` variable is never declared to be a `Point` type. However, TypeScript compares the shape of `point` to the shape of `Point` in the type-check. They have the same shape, so the code passes.

The shape-matching only requires a subset of the object’s fields to match.

```
const point3 = { x: 12, y: 26, z: 89 };  
logPoint(point3); // logs "12, 26"  
  
const rect = { x: 33, y: 3, width: 30, height: 80 };  
logPoint(rect); // logs "33, 3"  
  
const color = { hex: "#187ABF" };  
logPoint(color);
```

Argument of type '{ hex: string; }' is not assignable to parameter of type 'Point'.
Type '{ hex: string; }' is missing the following properties from type 'Point': x, y

There is no difference between how classes and objects conform to shapes:

```
class VirtualPoint {  
  x: number;  
  y: number;  
  
  constructor(x: number, y: number) {  
    this.x = x;  
    this.y = y;  
  }  
}  
  
const newVPoint = new VirtualPoint(13, 56);  
logPoint(newVPoint); // logs "13, 56"
```

If the object or class has all the required properties, TypeScript will say they match, regardless of the implementation details.

Next Steps

This was a brief overview of the syntax and tools used in everyday TypeScript. From here, you can:

- Read the full Handbook [from start to finish](#) (30m)
- Explore the [Playground examples](#)