10. TypeScript Assignability Quiz!

When delving into the world of TypeScript, one of the first features you hear about is its ***structural* type system**. Unlike other languages, you do not *have to* provide the exact type a function demands in TypeScript. Another type will do, as long as it's compatible with what's expected.

But what does it mean for a type to be ***compatible*** with another one?

Glad you asked! In this chapter, we'll embark on an exploration of TypeScript's concept of **assignability**. And what better way to sharpen our intuition than **playing a game**? Just like children disassembling their toys to understand them, we will try to understand TypeScript's most fundamental design choices by testing our assumptions and seeing what breaks. I'll test your intuition of assignability by asking you a series of questions, starting with the one right before you.

Let's find out if you're up to the challenge! 😅

```ts
// Is `"Welcome!"` assignable to `string`?

type Quiz = "Welcome!" extends string ? true : false;

// Click on either `true` or `false` to give your answer:

type Test = Expect<Equal<Quiz, true | false>>;
```

Assignability is the only language spoken by the type system. Whenever we get a type error, it's always a complaint about an assignability *rule* that has been broken. These rules may appear straightforward for primitive types, objects or arrays, but what about **unions**, **intersections**, **read-only** & **optional** properties, or even **function types**?

Contemplating these questions will take us to the strange land of **type *variance***, a concept that's often misunderstood and – even if you've never heard about it – has likely caused a stumble or two along your path!

Let's play

When it comes to quickly building an intuition, nothing beats a good game. Note that all the forthcoming questions presume that Strict Mode is enabled in our `tsconfig.json` (as it should be! 😈)

Let's get into it! 🎮

```ts
type Quiz = boolean extends true ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = "https" extends "http" ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = "3000" extends number ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = "1" | 0 extends number ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = "Hi!" extends string & number ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = "hello world!" extends `${string} ${string}`
  ? true
  : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = "8080" extends `${number}` ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = " > 1.618" extends `${string} > ${string}`
  ? true
  : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type User = { age: number; name: string };

type Quiz = { age: 21; name: "Amelia" } extends User ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Box = { value: number };
type Some<T> = { type: "Some"; value: T };

type Quiz = Some<123> extends Box ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type User = { age: number; name: string };

type Quiz = { name: "Gabriel" } extends User ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Post = {
  name: string;
  author: { id: string };
};

type Article = {
  name: string;
  author: { id: number };
};

type Quiz = Post extends Article ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Widget = {
  type: "barChart" | "lineChart" | "scatterPlot";
  position: { x: number; y: number; width: number; height: number };
};

const maybeWidget = {
  type: "barChart",
  position: { x: 0, y: 0, width: 100, height: 200 },
};

type Quiz = typeof maybeWidget extends Widget ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = number | string extends {} ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = number | string extends object ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = string[] extends (string | number)[] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = [1, 2, 3, 4] extends [number, number, number] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = [] extends string[] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = string[] extends [string, ...string[]] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = [] extends [string, ...string[]] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = number[] extends readonly number[] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = Readonly<{ prop: string }> extends { prop: string }
  ? true
  : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = [1, 2] extends number[] & { length: 2 } ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = [never] extends [1 | 2 | 3] ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = void extends never ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = { key: unknown } extends { key: string } ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = unknown extends null | {} | undefined ? true : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz = { key: unknown } extends { key: infer Key }
  ? true
  : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
type Quiz =
  42 extends infer MeaningOfLife extends string
    ? true
    : false;

type Test = Expect<Equal<Quiz, true | false>>;
```

```ts
function main<A, B>() {
  type Quiz = A extends B ? true : false;

  type Test = Expect<Equal<Quiz, true | false>>;
}
```

```ts
function main<A extends "a" | "b", B extends string>() {
  type Quiz = A extends B ? true : false;

  type Test = Expect<Equal<Quiz, true | false>>;
}
```

```ts
type Quiz =
  (() => "a" | "b") extends (() => string)
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```ts
type Quiz =
  (() => number) extends (() => 1 | 2)
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```ts
type Quiz =
  ((arg: number) => void) extends ((arg: 1 | 2) => void)
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

Function Types and the Upside Down

When you picture types as **sets of values**, assignability becomes much more intuitive. Well, at least until you start trying to understand the **assignability of function types**! They're significantly more challenging to visualize this way and the most effective approach to understanding their assignability is to imagine **functions taking callbacks**.

Let's turn one of our quiz questions into a concrete use case.

```ts
type Quiz =
  (() => "a" | "b") extends (() => string)
    ? true
    : false;
```

👇

```ts
async function createUser(getId: () => string) {
  await save({ type: "user", id: getId() });
  return "ok";
}
```

Imagine that Louise, an engineer on your team, wrote a `createUser` function that requires its caller to provide a way of generating an id. It takes a callback of type `() => string`, just like the second function in our quiz.

Marcel just joined the company and wants to create a user. He notices the following function in our codebase that returns either `"a"` or `"b"` at random:

```ts
function aOrB(): "a" | "b" {
  return Math.random() > 0.5 ? "a" : "b";
}
```

Here is the question: is Marcel allowed to pass `aOrB` to `createUser`?

```
createUser(aOrB); // does this type-check?
```

Well, `createUser` only requires its callback to return a string. `"a"` and `"b"` are strings, so this should type-check!

```
createUser(aOrB); // ✅
```

The rule is the following: a function of type `() ⇒ A` is assignable to a function of type `() ⇒ B` if `A` is assignable to `B`!

`() ⇒ A` is assignable to `() ⇒ B`
***if** `A` is assignable to `B`*

Nice 🎉

Now, Louise is a bit worried about people inadvertently giving the same id to two different users. She decides to fetch the **total count of users** in our database and **pass it to** `getId`, hoping people would create their id from this unique number:

```
declare function createUser(
  getId: (n: number) ⇒ string
  //       👆
): Promise<"ok">;
```

Marcel sees this, and decides to update `aOrB` to take it into account. Since it always returns either `"a"` or `"b"`, Marcel figures that it **doesn't *need*** the full `number` set as its input. *"Let's be **smart**"* he thinks, *"and make `aOrB` only take `1 | 2` instead of all numbers. Since `1 | 2` is assignable to `number`, this should work!"*

```
function aOrB(n: 1 | 2): "a" | "b" {
  return { 1: "a", 2: "b" }[n];
}

createUser(aOrB);
//         ~~~~ ❌
```

"You're making a terrible mistake, Marcel" whispers the type checker...

But this does ***not*** work. Marcel forgot something important: **he doesn't get to *choose*** what numbers are passed to `aOrB`. The `createUser` function can decide to pass any sort of number to its callback, not just `1` or `2`!

```
async function createUser(getId: (n: number) ⇒ string) {
  // `getId` can receive any number:
  const id = getId(1290921);
}
```

In other words, Since he doesn't ***provide*** this number but only **consumes** it, his code has to support **all numbers**.

Providers & Consumers

In the vast majority of the code we write, ***we*** provide values to things. We assign them to **variables**, pass them to **functions** or put them onto **objects**. For all of these cases, assignability works just the way we are used to — we are allowed to provide a **subtype** of what's expected, and everything works just fine:

```
// These are waiting for `number` values we should assign:
let data: number;
let obj: { value: number };
let fn = (data: number) ⇒ {};

// We can provide a subtype!
```

```
declare const value: 1 | 2 | 3;
data = value; // ✅
obj = { value }; // ✅
fn(value); // ✅
```

but passing a *supertype* is forbidden:

```
// We can't provide a supertype:
declare const value: number | string;

data = value; // ❌
obj = { value }; // ❌
fn(value); // ❌
```

In this code, we play the role of the provider because ***we*** assign these values. But when defining **function arguments**, things are quite different. Our code is **asking for values**. We are no longer providers, but **consumers**:

```
// These consume numbers:
let handler = (value: number) => {};
let obj: { fn: (value: number) => void };
let fnWithCallback = (cb: (value: number) => void) => {};

// We can't use subtyping in the argument position:
declare const fn: (value: 1 | 2 | 3) => void;

handler = fn; // ❌
obj = { fn }; // ❌
fnWithCallback(fn); // ❌
```

That makes sense because our `fn` function could be called with any number in these contexts. We can, however, update `fn` to take a **supertype** of `number` if we want to:

```
// We *can* use a supertype in the argument position:
declare const fn: (value: number | string) => void;

handler = fn; // ✅
obj = { fn }; // ✅
fnWithCallback(fn); // ✅
```

Of course, if `fn` **supports either numbers or strings**, we can use it in a context where it will only ever be given numbers! Here is the general **assignability rule** for function arguments:

> *`(arg: A) => T` is assignable to `(arg: B) => T` **if** `B` is assignable to `A`*

The **direction** of assignability is **inverted** for function arguments. It's the upside-down of assignability. In Type Theory lingo, people call this a ***contravariant* position**, in opposition to the standard way assignability works, called ***covariant***.

> *"Function arguments are a **contravariant** position. All other positions are **covariant**."*

Type ***variance*** refers to the direction of assignability. It isn't a property of a type — all types can be either covariant or contravariant — it's only a property of ***where*** this type is placed in a larger structure.

Variance & Type Parameters

You might be wondering why I'm insisting on the concept of type variance. Everything appeared quite intuitive when we considered callback functions, didn't it? So why do we need these unwieldy definitions?

Well, things start getting a lot less obvious when dealing with generic types. Naming these concepts will help us think about them! Take a look at this piece of code:

```ts
const httpPost = async (options: PostOptions<object>) => {
  // send an http request ...
};

const addUser = async (options: PostOptions<{ username: string }>) => {
  return httpPost(options);
  /*               👆
      Are we allowed to pass `options` to httpPost?
                                                    */
};
```

We are trying to assign a `PostOptions<{ username: string }>` to a `PostOptions<object>`. **Can we do that?**

🤔

Well, **it depends**!

If `PostOptions`'s parameter refers to a value ***we* provide**, then sure! `{username: string}` is assignable to `object` after all:

```ts
type PostOptions<T extends object> = {
  body: { data: T };
  /*           👆
           covariant
                        */
};
```

Here, the `T` parameter is placed in a **covariant** position — **an object property** — so this assignment type-checks:

```ts
const addUser = async (options: PostOptions<{ username: string }>) => {
  return httpPost(options);
  /*               👆 ✅
     `PostOptions<{ username: string }>`
     is assignable to `PostOptions<object>`.  */
};
```

But ***what if*** `PostOptions` was implemented this way instead:

```ts
type PostOptions<T extends object> = {
  onSuccess: (data: T) => void;
       /*           👆
              contravariant
                                */
};
```

Then `T` would refer to a value we **consume** — a **contravariant** position. In this case, subtyping the `object` type would no longer be allowed. We ***must*** provide a function that supports any object, so this assignment wouldn't type check:

```ts
const addUser = async (options: PostOptions<{ username: string }>) => {
  return httpPost(options);
  /*               👆 ❌
     `PostOptions<{ username: string }>`
     is *not* assignable to `PostOptions<object>`!  */
};
```

Generic types can sometimes trip you up. What may appear like **two identical pieces of code** can in fact have opposite type-checking behaviors!

Now that we know about variance, let's briefly go over the **two** remaining kinds: ***invariance*** and ***bivariance***.

Invariance

Invariance means that two instances of a generic type **can't be assigned in either direction**. It happens when a type parameter is both placed in a covariant ***and*** a contravariant position.

A common source of invariant types are generic UI components built with declarative frontend frameworks, like React or Vue:

```ts
type FormProps<T> = {
  value: T;
  //     👆 covariant
  onChange: (value: T) => void;
  //                👆 contravariant
};

declare const Input: (props: FormProps<string>) => JSX.Element;
```

Since the `T` parameter is used both as a property and inside a callback, subtyping breaks in both directions:

```ts
// `"a" | "b"` is a subtype of `string`.
const props: FormProps<"a" | "b"> = {
  value: "a",
  onChange: (value) => setState(value),
};

Input(props);
/*    ~~~~~
 *    ❌ Not assignable because `props.onChange` aren't.
 */
```

A supertype of `string` wouldn't work either:

```ts
// `string | number` is a supertype of `string`.
const props: FormProps<string | number> = {
  value: 32,
  onChange: (value) => setState(value),
};

Input(props);
/*    ~~~~~
 *    ❌ Not assignable because `props.value` aren't.
 */
```

Invariance is such a common source of typing issues in real-world codebases. Now that you get why subtyping is forbidden in such cases, I hope you will spend less time fighting the compiler!

Bivariance

Bivariance is the opposite of invariance. It means that **assignability works in both directions**. A very stupid example of a ***bivariant type*** is a generic type that doesn't use its parameter:

```ts
type Bivariant<T> = { "I'm": "bivariant!" };

declare let x: Bivariant<"👋">;
declare let y: Bivariant<string>;

x = y; // ✅
y = x; // ✅
```

In **Strict Mode**, bivariance isn't common[1] because type parameters are rarely left unused.

Explicit Variance with `in` and `out`

A weird TypeScript gotcha is that **generic classes** are **covariant** in their parameters by default, **even when they shouldn't be**. Here is an example:

```ts
class Box<T> {
  constructor(private value: T) {}

  set(value: T) {
    this.value = value;
  }
  get(): T {
    return this.value;
  }
}

const numberBox = new Box<number>(42);
numberBox.set(7);
numberBox.get(); // returns `7` of type `number`.
```

A `Box` contains an unknown value. It's type, `T`, is used both as a function argument ***and*** as a return type. This must mean this parameter is **invariant**, right?

Let's see what happens if we try re-assigning it to a different type:

```
// this type-checks ... 😳
let oopsBox: Box<number | string> = numberBox; // ✅
```

Hmm, looks like we **can** re-assign a `Box` to a wider type. The inferred variance is **wrong** here! This means we can run into the following kind of problems if we aren't careful:

```
oopsBox.set("Hello");
numberBox.get(); // returns "Hello" of type `number` 🙀🙀🙀
```

Since `oopsBox` points to the same value as `numberBox`, we can use it to set `numberBox`'s inner value to a string. This is a total **blindspot** in TypeScript's type system[2]! Luckily, there is a way to avoid this problem.

Since version 4.7, it's possible to explicitly annotate the variance of a generic type parameter using the `in` and the `out` keyword.

You can use the `out` keyword to define a ***covariant*** parameter:

```
type Covariant<out T> = { value: T };
//              👆

declare const x1: Covariant<"click" | "enter" | "leave">;

const x2: Covariant<string> = x1;
//    ~~ ✅
const x3: Covariant<"click"> = x1;
//    ~~ ❌
```

The `in` keyword to define a ***contravariant*** parameter:

```
type Contravariant<in T> = { onChange: (value: T) => void };
//                 👆

declare const x1: Contravariant<"click" | "enter" | "leave">;

const x2: Contravariant<string> = x1;
//    ~~ ❌
const x3: Contravariant<"click"> = x1;
//    ~~ ✅
```

Or both for an ***invariant*** parameter:

```
type Invariant<in out T> = { value: T; onChange: (value: T) => void };
//             👆 👆

declare const x1: Invariant<"click" | "enter" | "leave">;

const x2: Invariant<string> = x1;
//    ~~ ❌
const x3: Invariant<"click"> = x1;
//    ~~ ❌
```

As we've seen, TypeScript can accurately **infer** the variance of generic **types** and **interfaces**, but I strongly advise annotating the variance of class parameters explicitly. Let's make our `Box` class type-safe!

```
//       invariant
//          👇
class Box<in out T> {
  constructor(private value: T) {}

  set(value: T) { // 👈 contravariant
    this.value = value;
  }
  get(): T { // 👈 covariant
    return this.value;
  }
```

```
}

const numberBox = new Box(42);

const oopsBox: Box<number | string> = numberBox;
//    ~~~~~~~ ❌ 🎉

const oopsBox2: Box<1 | 2> = numberBox;
//    ~~~~~~~ ❌ 🙌
```

Cool.

That's enough with the theory, let's get back to our quiz!

Moar Quiz!

Let's see if you understand the assignability of function types better now 🙂

```
type Quiz =
  (args: (number | string)[]) => number
  extends
  (args: ("a" | "b" | "c")[]) => number
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```
type Quiz =
  ((value: unknown) => void) extends (value: never) => void
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```
type Quiz =
  ((id: symbol) => void) extends (value: unknown) => void
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```
type Quiz =
  (value: number) => number
  extends
  (value: number, index: number) => number
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```ts
interface Serializable<T> {
  serialize: (value: T) => string,
  deserialize: (str: string) => T | null,
};

type Quiz =
  Serializable<true> extends Serializable<boolean>
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```ts
type Pair<A, B> = { value: A, update: (value: B) => A };

type Quiz =
  Pair<1, unknown> extends Pair<number, string>
    ? true
    : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

```ts
type A = ((arg: { a: string }) => void)
       | ((arg: { b: number }) => void);

type B = (arg: { a: string } | { b: number }) => void;

type Quiz = A extends B ? true : false;

type Test = Expect<Equal<Quiz,  true  |  false  >>;
```

Summary 📚

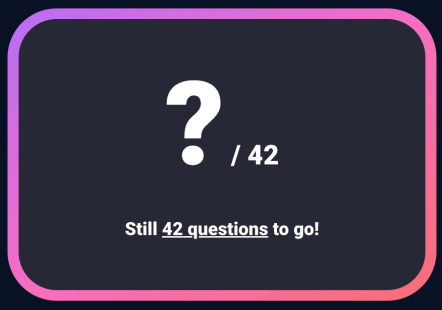
You've reached the end of the Assignability Quiz. **Congratulations!** 🎉 Brace yourself for a quick recap of the key concepts we've covered along the way. Let's dive in:

- If you want a **union** to be assignable to a certain type, every element within that union must be assignable to it.
- When it comes to assigning a type to an **intersection**, it must be assignable to all the intersected types.
- Remember that a `${string}` placeholder can be filled by the empty string in a **Template Literal Type**.
- An **object type** with extra keys can be assigned to an object type with fewer keys, as long as the shared key values match.
- The **empty object** type `{}` encompasses primitive types like `string` and `number`, but the `object` type does not!
- A **tuple type** can be assigned to another only if they have the **same length**.
- Everything can be assigned to `unknown`, but `unknown` can only be assigned to itself.
- `never` can be assigned to anything, but only itself can be assigned to `never`.
- The assignability of **function arguments** is inverted. It is called a ***contravariant*** position, in opposition to all other positions, called ***covariant***.

Type variance can sometimes be a source of confusion and frustration. It may seem arbitrary or even buggy if you don't grasp the underlying reasons why assignability behaves that way. I hope this

chapter has shed some light on this somewhat obscure concept. Now, it's your turn to explain it to your coworkers when they encounter a perplexing type error involving function types! 😅

And, of course, the moment you've been waiting for: **here's your final score**:



There will be no challenges for today, as this whole chapter has been its own set of challenges! I hope you learned a thing or two and had some fun along the way 🤗

Footnotes

1. **Without** the `strictFunctionTypes` option enabled in your `tsconfig.json` file, however, **all function arguments** will be considered **bivariant**! This behavior is totally unsound and can easily lead to runtime errors in production! I can't stress this enough: you should **always** enable strict type-checking on any TypeScript project that's used by real people. ↩

2. If the TypeScript team decided to always treat classes' type parameters as covariant, it was to decrease the number of type errors coming from mutable data structures, like the `Array` class. Array types are (unfortunately) mutable by default in TypeScript — which should make their inner type **invariant** — but invariance can be very annoying in practice since subtyping isn't allowed. Covariance would only be sound if arrays were **immutable**.

   In our declarative programming ecosystem though, we tend to treat JS arrays as immutable anyway, so I think giving up a bit of type soundness for some ease of use is an understandable tradeoff. ↩

⟸ Previous
9. Loops with Mapped Types

Next ⟹
11. Designing Type-Safe APIs