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>>;
```

Retry

That's right! 🎉

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, false>>;
```

Retry

Indeed 😎

`boolean` **isn't** assignable to `true`, but `true` **is** assignable to `boolean`!

The `boolean` **primitive** is actually defined as the union of the `true` and `false` **literals**:

```ts
type Test = Expect<Equal<boolean, true | false>>; // ✅
```

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

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

Retry

Good job 💪

Even though `"https"` starts with `"http"`, it doesn't mean it "extends" it!

I've always felt like the `extends` keyword was a little bit confusing, that's why I always rephrase it to *"is assignable to"* when reading it out loud. For two literal types like `"https"` or `"http"` to be assignable, they'd have to be equal.

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

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

Retry

Correct!

Even though it contains only digits, `"3000"` is a **string literal**, so it **isn't** assignable to `number`. Makes sense, imagine running this code:

```ts
const port = "3000";
const format = (num: number) => num.toFixed(2);

format(port); // 💥
```

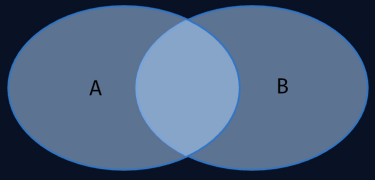
This would throw because the `toFixed` method only exists on numbers!

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

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

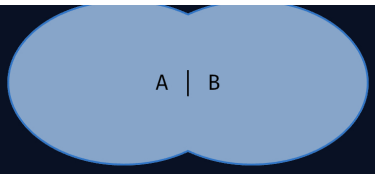
Retry

Yup 🎉

Types are **sets of values**, and a union type brings two of these sets together to form a larger set.



👇



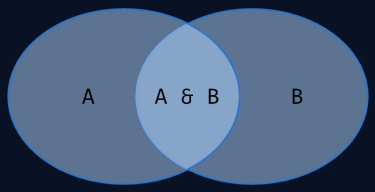
This means that for a union type `A | B` to be assignable to another type `C`, **both** `A` **and** `B` **have to be assignable to** `C`. Here, the string literal `"1"` **isn't** assignable to `number`!

If you were tempted to answer `true | false` because you remember from chapter 8 that conditional expressions distribute over union types, know that distributivity only happens when a type *parameter* is on the left-hand side of extends, not when a union is created inline (In other words, for "naked" type parameters).

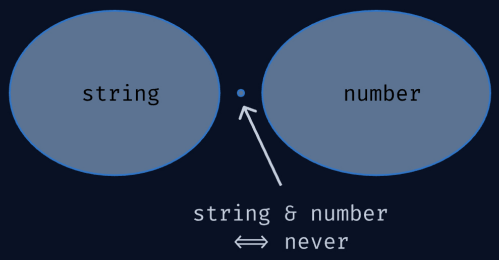
```
Retry
type Quiz = "Hi!" extends string & number ? true : false;

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

Good job 💪

Intersection types take the sets of two types, and only keep the subset that is assignable to both.



This case was kind of a trap, because `string & number` evaluates to `never`, the empty set. Nothing is assignable to `never`!



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

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

You're on a roll! 🛼

Template Literal Types define sets of strings that respect a certain **pattern**. `` `${string} ``

`${string}` Will match any string literal that contains a **space**, including `"hello world!"`, because `${string}` will match any possible string.

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

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

Retry

Correct!

A nice feature of Template Literal Types is their support for `number` and `boolean` slots in addition to `string`. They will match any string that represents values of these types. In this example, `` `${number}` `` will match any stringified number, but other strings wouldn't be assignable to it:

```
let numString: `${number}`;

numString = "1234"; // ✅
numString = "Oops"; // ❌
```

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

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

Retry

That's right!

With Template Literal Types, **the empty string can be assigned to a `${string}` slot**. The string `" > 1.618"` matches, because the first `${string}` slot matches `""`, and the second `${string}` slot matches the string `"1.618"`.

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

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

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

Retry

Yes!

`{ age: 21; name: "Amelia" }` is assignable to `User` because types contained in `age` and `name` are both assignable to the types in the corresponding properties of `User`:

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

declare const user1: { age: 21; name: "Amelia" };
const user2: User = user1; // ✅
```

```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>>;
```

Retry

You're killing it ✨

This one is a tiny bit more tricky. An object `A` can be assigned to an object `B` if it has **at least** all properties of `B`, and they all contain assignable types. `Some<123>` evaluates to `{type: "Some"; value: 123}`, which is assignable to `{value: number}` because `123` is assignable to `number`:

```ts
declare const some: { type: "Some"; value: 123 };

const box: { value: number } = some; // ✅

box.value;
//    ^? number
```

When assigning `some` to `box`, we essentially *forget* about extra properties that `some` had. We will no longer be able to access them from the `box` variable.

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

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

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

Retry

Correct!

`{ name: "Gabriel" }` isn't a `User` because it's missing an `age` property. The main takeaway here is that objects with ***fewer* properties** are the **supersets** of objects with ***more* properties**, not the other way around!

```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, false>>;
```

Retry

Correct 💪

Assignability checks are **recursive**, so if your data structures contain nested objects, matching properties on any depth must be assignable. Here, `Post["author"]["id"]` and `Article["author"]["id"]` are incompatible because `string` isn't assignable to `number`.

```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, false>>;
```

Retry

You have a sharp eye! 🧗

When creating an object literal with `{ ... }`, TypeScript ***infers*** its type for us.

But here is the thing: properties will be inferred as **primitive types** instead of literal types by default! In this case, the `type` property of `maybeWidget` is inferred as a `string`, which isn't assignable to `"barChart" | "lineChart" | "scatterPlot"`.

We can circumvent this by annotating `"barChart"` with `as const`, and tell TypeScript we want this property to be inferred as a literal type:

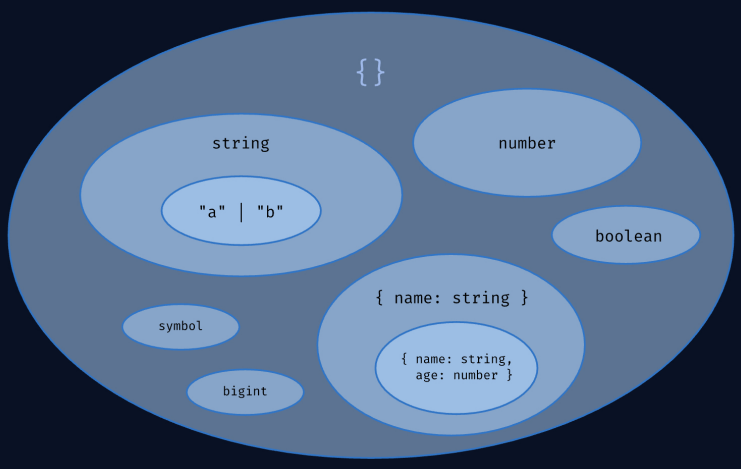
```ts
const maybeWidget = {
  type: "barChart" as const, // 👈
  position: { x: 0, y: 0, width: 100, height: 200 },
};

const widget: Widget = maybeWidget; // ✅
```

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

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

Retry

Nice job!

The empty object `{}` type defines **one of the largest set of values** in TypeScript. All types with properties or methods are assignable to it, including `number` and `string`!



The empty object set.

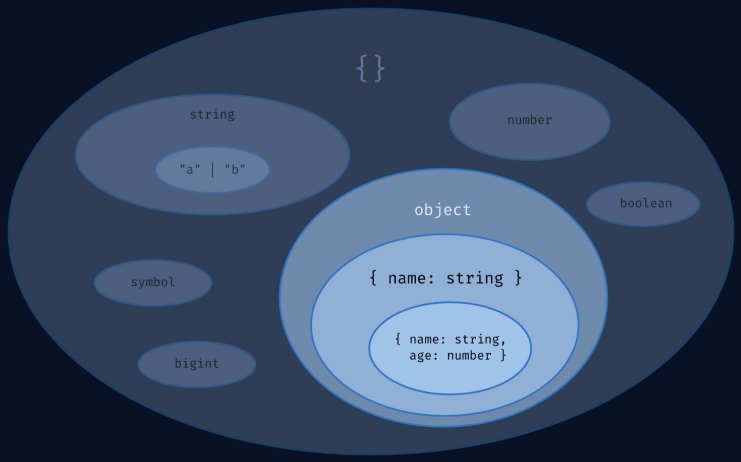
```ts
type Quiz = number | string extends object ? true : false;

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

Retry

Indeed 🤔 Looks like you know all the little tricks of TypeScript's assignability!

I used to think that `object` was nothing more than an alias for `{}`, but I was wrong! **Here is where `{}` and the `object` type differ**:

- `{}` includes **primitive types**
- but `object` does ***not***!



The `object` set.

If you are not convinced, try editing this little playground:

```ts
let a: {};
a = 12; // ✅
a = "😎"; // ✅

let b: object;
b = 403; // ❌
b = "⛑"; // ❌
b = { some: "object" }; // ✅
```

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

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

Retry

That's right 🙂

For an array type to be assignable to another one, **their inner types only need to be assignable too**. Here, `string` is assignable to `string | number`, so this works.

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

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

Retry

You're crushing it.

For a tuple to be assignable to another one, not only do they need to contain assignable types for every index, but **their lengths must match as well**!

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

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

Retry

Awesome work 🤩

The empty tuple `[]` type can be assigned to any array. You can think about it as an array containing `never`.

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

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

Retry

You're good!

`[string, ...string[]]` is the type of **non-empty lists** of strings. It uses a Variadic Tuple to tell TypeScript that a string is always present at the first position. `string[]` isn't assignable to it, because it may very well be empty!

```ts
const strings: string[] = [];

const first = <T>(list: [T, ...T[]]): T => list[0];

first(strings);
//    ~~~~~~~ ❌
```

If `first` could take any arrays, we would need to change its return type to `T | undefined`, but here this is safe!

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

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

Retry

This one is pretty intuitive. An empty array is just ***not*** assignable to a non-empty array!

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

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

Retry

Correct!

Read-only arrays cannot be mutated in any way. You aren't allowed to use some of the most common array methods on them because they'd change their content:

```ts
declare const nums: readonly number[];

nums.push(2);
//   ~~~~ ❌ adds a value to `nums`!
nums.pop();
//   ~~~  ❌ removes the last element!
nums.sort();
//   ~~~~ ❌ changes the ordering!
```

`readonly` is particularly useful when an array *reference* is **shared** between **several function scopes**. You typically never want to mutate arrays received as arguments because doing so is likely to introduce hard-to-find **bugs**. Here is a silly example:

```ts
async function doThings(friends: User[]) {
  const users = [currentUser, ...friends];

  await sayHi(users);

  // Oh, no 🙀 users[0] is no longer currentUser:
  await grantAdminPriviledges(users[0]);
}

async function sayHi(users: User[]) {
  // .shift() removes the first element
  // from the array 🙈
  await sendMessage(users.shift(), "Hello!");
}
```

Annotating an array argument with `readonly` only means that ***this function*** isn't allowed to mutate it. It's however absolutely fine to pass mutable arrays to it:

```ts
const fn = (arr: readonly number[]) => {
  // can't mutate `arr` in here!
  return arr.length;
};

let arr: number[] = [];
arr.push(1);

fn(arr); // ✅
```

That is why regular arrays are assignable to `readonly` ones. 🙂

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

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

Retry

That's unfortunately correct! 🙃

Given what we just saw regarding read-only arrays, wouldn't it be natural for read-only objects ***not*** to be assignable to mutable ones?

Well, yeah! If we want our object to be ***immutable***, we surely don't want to be allowed to pass it to a function that can mutate it.

But it turns out there is a bug in the current version of TypeScript that let's you assign read-only objects to mutable objects. Here is a link to the relevant issue on TypeScript's GitHub repository. Make sure to give it a 👍 if, like me, you would like to see this fixed!

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

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

Retry

That's indeed true! 👏

When an intersection type like `A & B` is present on the right-hand side of `extends`, the type on the left has to be assignable to both `A` and `B`.

In this case `[1, 2]` is indeed assignable to an array of numbers. It also has a `"length"` property equal to the literal type `2`, so `{ length: 2 }` matches as well.

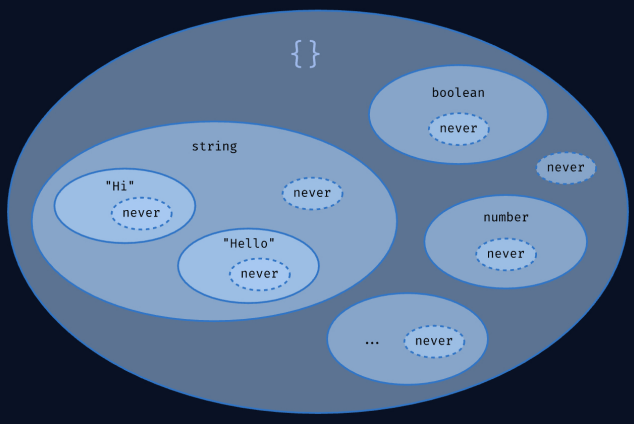
`number[] & { length: 2 }` is kind of a pedantic version of the `[number, number]` type. 😅

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

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

Retry

Easy as `1 | 2 | 3` 🍥

`never` is the only type that doesn't represent any runtime value. If you think in terms of Set Theory, `never` is the **empty set** — Nothing is assignable to it. It also means that `never` is the **subtype** of **every other type**!



There's always some room for `never`.

Since `never` is assignable to `1 | 2 | 3`, `[never]` has to be assignable to `[1 | 2 | 3]`!

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

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

Retry

You're on 🔥

Nothing is assignable to `never`, not even `void`!

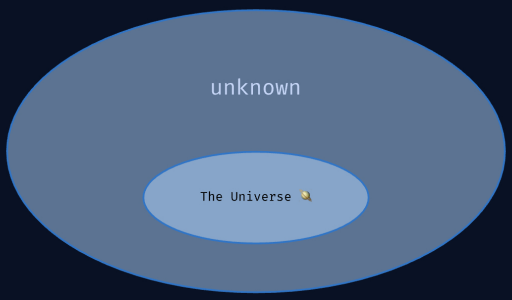
`void` is the **return type** that **TypeScript infers** for **functions without a** `return` **statement**. I tend to think of `void` as another name for `undefined`. This is technically not true, because `void` and `undefined` have some slight differences in behavior, but they are so subtle that I don't think they're worth keeping in mind. I kind of wish TypeScript only had an `undefined` type.

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

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

Retry

Yes! 🙌

`unknown` is the **superset** of every type that you'll ever be able to create with TypeScript. It contains everything else, so it **can't be assigned** to anything except itself!



The known universe is small compared to the unknown... ✨

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

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

Retry

Hats off 🙇

I just said that `unknown` could **only** be assigned to **itself**, so what's going on here, ***was I lying?!*** 😱

I'd `never` do that! 😅 It's simply that `unknown` is just an alias for the following **union type**:

```ts
type unknown = null | {} | undefined;
```

The only types that **aren't** part of the `{}` set are `null` and `undefined`. Put them all in a union and you get `unknown`! That's why this test passes 😊

It also explains why the `NonNullable` built-in helper works:

```ts
// `NonNullable`, as defined in TypeScript's standard library:
type NonNullable<T> = T & {};

type T1 = NonNullable<unknown>; // {}
type T2 = NonNullable<number | undefined>; // number
```

By intersecting any type with `{}`, we are filtering `null` and `undefined` out!

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

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

Retry

Correct 🎉

The `infer` keyword lets us **extract** a type from a data structure by assigning it to a **variable**. In this example, we assign the type of the `key` property to a `Key` variable. We could have used `Key` in the truthy code branch, but we didn't.

From an assignability perspective, `infer` will match any type, just like `any`, so `unknown` is assignable to `infer Key`!

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

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

Retry

Good job 👍

When declaring a variable with `infer`, we can give it a **type constraint** by using the `extends` keyword a second time. The condition will only pass if the assigned type respects the constraint. Here it isn't the case because `42` isn't assignable to `string`!

```ts
function main<A, B>() {

  type Quiz = A extends B ? true : false;

  type Test = Expect<Equal<Quiz, true>>;

}
```

Retry

Well, there is no correct answer here. 🤭

In TypeScript, you can create types anywhere, including in the body of a generic function. Here, since the type checker doesn't know what type `A` and `B` may contain, it **can't check if one is assignable to the other!**

When this happens, the conditional expression **does not *reduce*** — the type of `Quiz` remains as it is written: `A extends B ? true : false`.

And don't worry, I'll still count your answer as correct in your final score 😉

```
function main<A extends "a" | "b", B extends string>() {

  type Quiz = A extends B ? true : false;

  type Test = Expect<Equal<Quiz, false>>;

}
```

Retry

TypeScript got stuck again 🤯

Since `A` must be assignable to `"a" | "b"` and `B` must be assignable to `string`, you could be tempted to think that `A` must be assignable to `B`, but that's not true! All we know is that `B` must be inside the `string` set, but it could very well be another literal type than `"a"` and `"b"`, like `"banana"` or `"pineapple juice 🍍"`!

That's why TypeScript stops ***reducing*** this type. It will remain `A extends B ? true : false` forever.

Even though it makes sense, this behavior is sometimes frustrating because it may cause type errors that **can't be fixed** without resorting to unsafe escape hatches. If this rings a bell and you'd like to know more, I wrote a piece about what I consider the best approach to Make Generic Functions Pass Type Checking. Check it out!

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

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

Retry

Correct ✨

That works because `"a" | "b"` is assignable to `string`!

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

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

Retry

Amazing!

Since `number` contains other values than only `1` or `2`, A `() => number` function isn't assignable to `() => 1 | 2`.

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

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

Retry

Great job! 🔥

Wait a second. If `(arg: number) => void` is assignable to `(arg: 1 | 2) => void`, doesn't it mean that `number` should be assignable to `1 | 2`? But we *know* this isn't true! What's going on here?

Building an accurate intuition of **the assignability of function types** is significantly more challenging than the quiz we've been playing so far. Passing functions around is such a common TypeScript pattern that I propose we pause our little game for a minute and switch back to full-fledged chapter mode to dive deeper into this topic and make sure our **mental model** of function types is solid!

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`?

```ts
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!

```ts
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
// `"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:

```
// `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:

```
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:

```
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 😊

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

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

Exactly!

The direction of assignability is inverted for function arguments. Since `("a" | "b" | "c")[]` is assignable to `(number | string)[]`, this test passes!

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

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

Good job!

Nothing is assignable to `never`. Well, ***except* in a contravariant position**! Since the assignability is inverted, every type becomes assignable to it, including `unknown`. A function that takes `never` basically means that we aren't allowed to call it at runtime, so we don't really care if the assigned function used to take a string or a potato.

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

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

Yoohoo! 🎉

When `unknown` is placed in a contravariant position, no other type can be assigned to it. This is something to keep in mind when creating generic *function constraints*. **Can you see what's wrong with the following function?**

```ts
function call<F extends (value: unknown) => unknown>(
  fn: F,
  arg: Parameters<F>[0]
) { /* ... */ }
```

Since no argument type can be assigned to an `unknown` argument, we won't be allowed to pass any function to `call`:

```
call((x: number) => x + 1, 2);
/*     ~~~~~~~~~~~~~~~~~~~~
       ❌ `unknown` isn't assignable to `number`.    */

call((x: string) => `${x}!`, "Hello");
/*     ~~~~~~~~~~~~~~~~~~~~
       ❌ `unknown` isn't assignable to `string`.    */
```

But it would work with `never`:

```
function call<F extends (value: never) => unknown>(
  fn: F,   //                    👆
  arg: Parameters<F>[0]
) { /* ... */ }

call((x: number) => x + 1, 2);
/*    ^ ✅ `never` *is* assignable to `number`!    */

call((x: string) => `${x}!`, "Hello");
/*    ^ ✅ `never` is also assignable to `string`!    */
```

To avoid thinking about variance too much, I tend to use `any` rather than `never` or `unknown` in type constraints. Since `any` is a *subtype* and a *supertype* of every type, I know the direction of assignability won't trip me up!

```
function call<F extends (value: any) => any>( ... );
//                           👆      👆
```

Using `any` **inside a type *constraint*** is fine because no value will end up being of type `any`. TypeScript will infer a more precise type from the argument given at each call site!

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

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

Retry

You're crushing it!

Functions taking ***fewer*** arguments **are assignable** to functions taking ***more*** arguments.

That's good news because, in JavaScript, we frequently pass too many arguments without realizing it. Without this rule, the following code wouldn't type-check:

```
const add1 = (n: number) => n + 1;
const result = [1, 2, 3].map(add1); // [2, 3, 4]
```

Indeed, `map` not only passes the current value to its callback, but also the **current index** as well as the **full array**. In this particular instance, the `map` method takes the following type:

```
function map(
  mapperFn: (
    value: number,
    index: number,
    array: number[]
  ) => number
): number[];
```

But that's not a problem because `add1` is assignable to `mapperFn`!

```
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, false>>;
```

Retry

That's right! 👌

Generic types that can be serialized and deserialized are the perfect example of ***invariance***. Since the parameter is both positioned as covariant and contravariant, there is no way it can support subtyping. That's why `Serializable<true>` ***isn't*** assignable to `Serializable<boolean>`.

```
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>>;
```

Retry

Yes!

The first parameter of the `Pair` generic is covariant, and the second one is ***contravariant*** because it refers to the type of an argument! `1` is indeed assignable to `number`, and `string` is assignable to `unknown`, so TypeScript is ok with this.

```
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, false>>;
```

Retry

Wow, you're unstoppable! 🏃

We are trying to assign a **union of functions** to a function taking the **union** of their **parameters**. This case is pretty tricky to reason about abstractly, so let's turn it into functions and callbacks:

```
const main = (
  callback: (arg: { a: string } | { b: number }) => void
) => {
```

```
  ...
};
```

Our `main` function can call its callback with either a `{ a: string }` or a `{ b: number }`:

```
const main = (callback: ...) => {
  callback({ a: "hi" }); // ✅
  callback({ b: 12 }); // ✅
};
```

`callback` has to support these two inputs!

A union of functions represents the fact that we could either have one or the other at runtime. Both functions should therefore be assignable for the overall union type to be assignable.

Is it the case here?

```
declare const takesA: (arg: { a: string }) => void;
declare const takesB: (arg: { b: number }) => void;

main(takesA);
/*   ~~~~~~
     ❌ Can't assign `{ a: string } | { b: number }`
        to `{ a: string }`.                          */
main(takesB);
/*   ~~~~~~
     ❌ Can't assign `{ a: string } | { b: number }`
        to `{ b: number }`.                          */
```

It isn't! Neither of our functions supports the full set of inputs that `main` can provide. This assignability check fails!

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**:

9 / 42

correct answers 🎉

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