The `unknown` Type

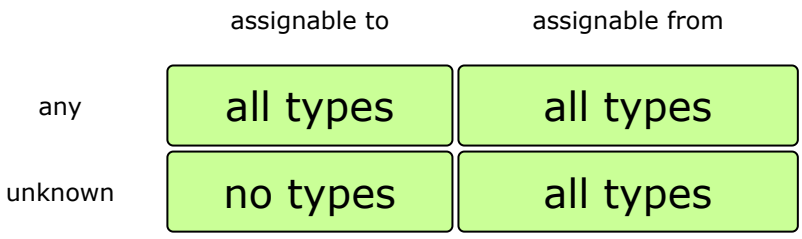
This lesson explains the `unknown` type and explains how it is a better alternative to using `any`.

WE'LL COVER THE FOLLOWING

Overview

When trying to eliminate `any` from your codebase, it's useful to know about the `unknown` type. It is a safer alternative to `any`. Both `any` and `unknown` represent an unknown type. However, there is a key difference between these two:

- all types are assignable to the `any` type and the `any` type is assignable to any other type
- all types are assignable to the `unknown` type, but the `unknown` type is not assignable to any type



Type assignability

What does it mean when a type is assignable to another type? We say that

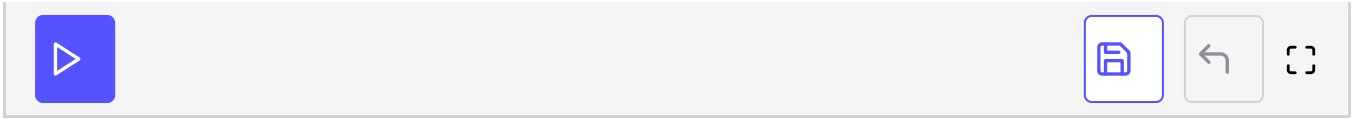
type `A` is assignable to type `B` if you can use a value of type `A` in all places where a value of type `B` is required.
Here are some examples of situations when type assignability is checked:

- assigning a value to a variable
- passing a function argument
- returning a value from a function that has an explicit return type

As mentioned, the key difference between `any` and `unknown` is that `any` is assignable to any other type, but `unknown` is not assignable to any type. Let's look at some examples.

```
function foo1(bar: any) {
  const a: string = bar; // no error
  const b: number = bar; // no error
  const c: { name: string } = bar; // no error
}

function foo2(bar: unknown) {
  const a: string = bar; // 🔘 Type 'unknown' is not assignable to type 'string'.(2322)
  const b: number = bar; // 🔘 Type 'unknown' is not assignable to type 'number'.(2322)
  const c: { name: string } = bar; // 🔘 Type 'unknown' is not assignable to type '{ name: s
}
```

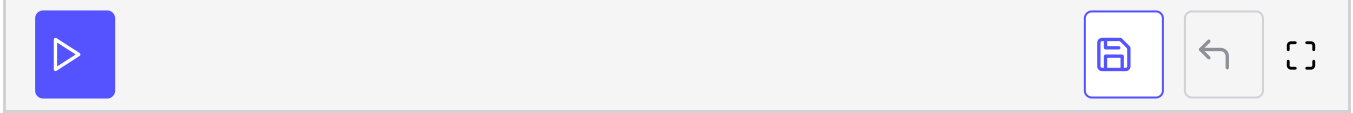


Run the code to see the compile errors.

On the other hand, all types are assignable to `unknown`.

```
let x: unknown;

x = 123; // no error
x = 'abc';  // no error
x = true; // no error
```



Run the code to see that there are no errors.

Why is this difference so important? If you have a value of `unknown` type, you need to cast it to some other type before you can do anything useful with it. The consequence of this is that `unknown` doesn't *propagate* like `any` does, which is much safer.

Usage example for `unknown` type

When trying to get rid of the `any` instances from your codebase, you might run into situations where you actually have no way of knowing the type of some value. This is especially true when the value is "external", meaning it's returned by a backend endpoint or is deserialized from local storage. In such cases, it's a good idea to type such value as `unknown` instead of `any`.

```
interface Article {
    title: string;
    body: string;
}

fetch("http://example.com")
  .then(response => response.json())
  .then((body: unknown) => {
    const article = body as Article;
    // we need to cast, otherwise we'd get an error
    console.log(article.title);
  });
```

In this example, we use type assertion to tell TypeScript that we know the type of `body`. It's still not type-safe because we could be wrong, but at least it's explicit. The proper solution here would be to perform a runtime check to make sure that `body` is indeed an `Article`. We'll look into such a solution in the lesson dedicated to user-defined type guards.

No explicit `any`

With the `unknown` type, it's possible to completely avoid having `any` in your codebase. While there is no compiler flag to enforce the absence of `any`, you might consider using the `no-explicit-any` ESLint rule for this purpose. You can find the rule here.

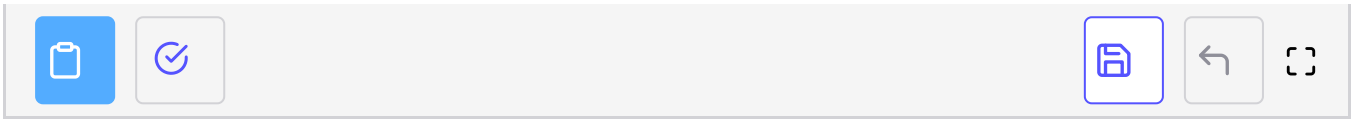
ESLint is the code linter (a tool that analyzes code for errors, bugs, or stylistic inconsistencies) for JavaScript and TypeScript. TypeScript used to have a dedicated linter called TSLint, but it was deprecated in favor of ESLint. ESLint is used with the `eslint-plugin-typescript` plugin. I highly recommend looking into ESLint as it may help make your types even more strict.

Exercise

Write a runtime check that could be used in the above code to make sure that

`body` is indeed an `Article`.

```
function isArticle(body: unknown): boolean {
    return null;
}
```



In the following lesson, we'll start discussing the most important compiler flag related to strictness, the `strictNullChecks` flag.