TypeScript Unknown Type

Summary: in this tutorial, you will learn about the TypeScript unknown type to enforce type checking of an unknown value.

Introduction to the TypeScript unknown type

In TypeScript, the `unknown` type can hold a value that is not known upfront but requires type checking.

To declare a variable of the `unknown` type, you use the following syntax:

```
let result: unknown;
```

Like the any type, you can assign any value to a variable of the `unknown` type. For example:

```
let result: unknown;

result = 1;
result = 'hello';
result = false;
result = Symbol();
result = { name: 'John' };
result = [1, 2, 3];
```

Unlike the `any` type, TypeScript checks the type before performing operations on it. For example, you cannot call a method or apply an operator on a `unknown` value. If you attempt to do so, the TypeScript compiler will issue an error:

```
let result: unknown;
result = [1,2,3];

const total = result.reduce((a: number, b:number ) => a + b, 0);
console.log(total);
```

In this example, the `result` variable has the type of `unknown`. We assign an array the `result` value, but its type is still `unknown`. Therefore, we cannot call the `reduce()` method of an array on it.

To call the `reduce()` method on the `result` variable, you need to use the type assertion to explicitly tell the TypeScript compiler that the type of the result is array. For example:

```
let result: unknown;
result = [1, 2, 3];

const total = (result as number[]).reduce((a: number, b: number) => a + b, 0);
console.log(total); // 6
```

In this example, we explicitly tell the TypeScript compiler that the type of the `result` is an `array` of numbers (`result as number[]`).

Therefore, we can call the `reduce()` method on the `result` array without any issues.

Unknown vs Any type

The following table highlights the key differences between the `unknown` and `any` types:

Feature	any	unknown
Type Safety	No type-safety	Enforces type safety
Operations	Operations can be performed without checks	Operations cannot be performed without type assertion (narrowing type)
Use cases	Useful for dynamic values but unsafe.	Useful for dynamic values and safe because it requires validation before use.

Feature	any	unknown
Type Checking	TypeScript compiler does not perform a type checking on an any variable.	TypeScript compiler enforces a type checking on an unknown variable.
Common Scenarios	Used for migrating JavaScript codebase to TypeScript.	Used when handling data from external sources (API calls, databases, ..) where type validation is necessary.

TypeScript unknown examples

Let's take some practical examples of using the Typescript `unknown` type.

1) Handling external data

When receiving data from an external API, you can use the `unknown` type to enforce validation before processing it.

The following example shows how to use the fetch method to call an API from the `https://jsonplaceholder.typicode.com/posts` endpoint:

```
const fetchData = async (url: string): Promise<unknown> => {
  const response = await fetch(url);
  return await response.json();
};

const showPosts = async () => {
  const url = 'https://jsonplaceholder.typicode.com/posts';
  try {
    const posts = await fetchData(url); // unknown type

    (
      posts as { userId: number; id: number; title: string; body: string }[]
    ).map((post) => console.log(post.title));
  } catch (err) {
    console.log(err);
  }
};
```

```
showPosts();
```

How it works.

First, define a function `fetchData` that calls API from a URL and returns JSON data. Since the shape of the returned data is not known, the function returns a `Promise<unknown>` value:

```
const fetchData = async (url: string): Promise<unknown> => {
  const response = await fetch(url);
  return await response.json();
};
```

Second, define the `showPosts()` function that uses the `fetchData()` function to call an API from the endpoint `https://jsonplaceholder.typicode.com/posts`:

```
const showPosts = async () => {
  const url = 'https://jsonplaceholder.typicode.com/posts';
  try {
    const posts = await fetchData(url); // unknown type
    (
      posts as { userId: number; id: number; title: string; body: string }[]
    ).map((post) => console.log(post.title));
  } catch (err) {
    console.log(err);
  }
};
```

In this example, the posts variable has a type of unknown.

Before accessing its `title` property, we use type assertion to instruct the TypeScript compiler to treat it as an array of post objects:

```
posts as { userId: number; id: number; title: string; body: string }[]
```

Third, call the `showPosts()` function:

```
showPosts();
```

2) Creating type-safe interfaces

The following example defines a function `format` that format a value before logging it to the console:

```
function format(value: unknown): void {
  switch (typeof value) {
    case 'string':
      console.log('String:', value.toUpperCase());
      break;
    case 'number':
      console.log('Number:', value.toFixed(2));
      break;
    default:
      console.log('Other types:', value);
  }
}
```

In this example, before accessing a method of the value, we validate its type to ensure that the operation is valid.

Summary

- The `unknown` type is like any type but more restrictive.
- Use the `unknown` type to handle data coming from external sources and requires validation before use.