Custom Type Guards

This lesson explains custom type guards and how they're useful.

WE'LL COVER THE FOLLOWING

Overview

TypeScript lets you define your own type guards. A custom type guard is a Boolean-returning function that can additionally assert something about the type of its parameter. If the function is called in the context of a conditional expression, the type of the passed value is narrowed to the asserted type inside the positive branch of the conditional.

Below, you can see an example of an `isArticle` type guard. It's a function that takes an object and checks if it's an instance of `Article` by checking for the presence of certain properties. Later, when the function is used inside the `if` statement, the type of `body` is narrowed to `Article` in the first branch of the conditional statement.

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

function isArticle(object: any): object is Article {
  return "title" in object && "content" in object;
}

fetch("http://example.com")
  .then(response => response.json())
  .then(body => {
    if (isArticle(body)) {
      return body.title;
    } else {
      throw new Error("This is not an article");
```

```
  }
});
```

Hover over `body` on line 14 to see how it's type is narrowed.

Runtime checks

Custom type guards are often used when dealing with data coming from the outside world. TypeScript doesn't have any runtime checks built-in. For example, if we fetch some data from the backend and assert its type as `Article` then TypeScript has no way to actually verify it, it has to believe us. A custom type guard is a safer alternative to type assertion (a cast) as it allows us to perform a runtime check before assertion. However, the correctness of the runtime check is our responsibility; if the check is not exhaustive then we may end up with a runtime error anyway.

For example, in the above piece, we don't check that the type of the `title` property is `string`. The backend might return an object that has the `title` property with the value `42`. Later, we might try to call the `toLowerCase()` method on the `body.title` because we think it's a string. At this point, we would get a runtime error.

In summary, type guards are like customizable type assertions where you can perform a runtime check to ensure the validity of the assertion. It's up to you to write the check correctly and to ensure that it's sufficient.

The final lesson introduces one more advanced type concept, nominal types.