```
  return jsonPromise;
}
```

Things to Remember

- Prefer Promises to callbacks for better composability and type flow.
- Prefer `async` and `await` to raw Promises when possible. They produce more concise, straightforward code and eliminate whole classes of errors.
- If a function returns a Promise, declare it `async`.

Item 26: Understand How Context Is Used in Type Inference

TypeScript doesn't just infer types based on values. It also considers the context in which the value occurs. This usually works well but can sometimes lead to surprises. Understanding how context is used in type inference will help you identify and work around these surprises when they do occur.

In JavaScript you can factor an expression out into a constant without changing the behavior of your code (so long as you don't alter execution order). In other words, these two statements are equivalent:

```
// Inline form
setLanguage('JavaScript');

// Reference form
let language = 'JavaScript';
setLanguage(language);
```

In TypeScript, this refactor still works:

```
function setLanguage(language: string) { /* ... */ }

setLanguage('JavaScript');  // OK

let language = 'JavaScript';
setLanguage(language);  // OK
```

Now suppose you take to heart the advice of Item 33 and replace the string type with a more precise union of string literal types:

```
type Language = 'JavaScript' | 'TypeScript' | 'Python';
function setLanguage(language: Language) { /* ... */ }

setLanguage('JavaScript');  // OK

let language = 'JavaScript';
setLanguage(language);
```

```
         // ~~~~~~~~ Argument of type 'string' is not assignable
         //          to parameter of type 'Language'
```

What went wrong? With the inline form, TypeScript knows from the function declaration that the parameter is supposed to be of type `Language`. The string literal `'JavaScript'` is assignable to this type, so this is OK. But when you factor out a variable, TypeScript must infer its type at the time of assignment. In this case it infers `string`, which is not assignable to `Language`. Hence the error.

(Some languages are able to infer types for variables based on their eventual usage. But this can also be confusing. Anders Hejlsberg, the creator of TypeScript, refers to it as "spooky action at a distance." By and large, TypeScript determines the type of a variable when it is first introduced. For a notable exception to this rule, see Item 41.)

There are two good ways to solve this problem. One is to constrain the possible values of `language` with a type declaration:

```
let language: Language = 'JavaScript';
setLanguage(language);  // OK
```

This also has the benefit of flagging an error if there's a typo in the language—for example `'Typescript'` (it should be a capital "S").

The other solution is to make the variable constant:

```
const language = 'JavaScript';
setLanguage(language);  // OK
```

By using `const`, we've told the type checker that this variable cannot change. So TypeScript can infer a more precise type for `language`, the string literal type `"JavaScript"`. This is assignable to `Language` so the code type checks. Of course, if you do need to reassign `language`, then you'll need to use the type declaration. (For more on this, see Item 21.)

The fundamental issue here is that we've separated the value from the context in which it's used. Sometimes this is OK, but often it is not. The rest of this item walks through a few cases where this loss of context can cause errors and shows you how to fix them.

Tuple Types

In addition to string literal types, problems can come up with tuple types. Suppose you're working with a map visualization that lets you programmatically pan the map:

```
// Parameter is a (latitude, longitude) pair.
function panTo(where: [number, number]) { /* ... */ }

panTo([10, 20]);  // OK

const loc = [10, 20];
```