Optionality

This lesson discusses how TypeScript analyses optional values in `if` statements when the `strictNullChecks` flag is enabled.

WE'LL COVER THE FOLLOWING

- Defining optionality
- Checking optionality
- Exercise

Defining optionality

As you saw in the previous lesson, strict null checks force you to explicitly distinguish between values that can be `null` or `undefined` and those that cannot be. You already saw how to do this with a union type.

```
interface Person {
  name: string;
}

let nullableJohn: Person | null;
let maybeUndefinedBob: Person | undefined;
let ambiguoslyEmptyAlice: Person | null | undefined;
```

There is another way to express optionality for function parameters and object properties.

```
interface Person {}

function hello(person?: Person) {}

interface Employee {
  id: number;
  name: string;
  supervisorId?: number;
}
```

```
hello();
const employee: Employee = {
  id: 1,
  name: 'John',
};
```

Run the code to see that there are no compile errors even though `person` argument and `supervisorId` property are skipped (`strictNullChecks` flag enabled).

The above definitions are the same as the ones below with one subtle difference.

```
function hello(person: Person | undefined) {}

interface Employee {
  id: number;
  name: string;
  supervisorId: number | undefined;
}
```

With top definitions, you can skip the function argument/object property altogether while with the bottom ones you need to explicitly provide them, even when they are `undefined`.

```
interface Person {}

function hello(person: Person | undefined) {}

hello();
```

Run the code to see how this time skipping `person` is not allowed. Modify the code to get it to compile (`strictNullChecks` flag is enabled).

It sometimes makes sense to use the latter form, as it might better reflect your intention. It ensures that the caller doesn't forget about some property or function argument.

Checking optionality

As previously mentioned, before you can do anything with an optional value

As previously mentioned, before you can do anything with an optional value, you need to check if it's empty.

```
interface Person { hello(): void }

function sayHello(person: Person | undefined) {
  person.hello(); // ▨ Error!

  if (person !== undefined) {
    person.hello(); // OK
  }
}
```

Run the code to see how line 4 produces an error (`strictNullChecks` flag is enabled).

TypeScript's type checker is clever enough to analyze the condition of the `if` expression and deduce that the type of `person` inside the `if` statement's body is narrowed to just `Person`. This is an example of a *type guard*, a concept that we will discuss in subsequent lessons.

The following variant is a common pattern to quickly check optionality.

```
if (person) {
  person.hello();
}
```

Most of the time it's fine, however, you need to be careful when using it with primitive types.

```
function sayHello(name?: string) {
  if (name) {
    console.log(`Name: ${name.toLowerCase()}`);
  }
}

sayHello('');
```

Run the code to see that it doesn't print anything, even though one might expect it to print "Name:" (`strictNullChecks` flag enabled).

This code won't print anything even though the provided value is defined. `if (name)` will check whether `name` is *falsy*. An empty string is a falsy value, so `(name)` will check whether `name` is *falsy*. An empty string is a falsy value, so

the condition will evaluate to `false`. It might be fine in this case, but in general, it’s safer to explicitly compare with `null` or `undefined`.

Exercise

Change the definition of `getArticleAuthorName` so that the code compiles with `strictNullChecks` enabled.

```
interface Author {
    name: string;
}

interface Article {
    title: string;
    author?: Author;
}

function getArticleAuthorName(article: Article) {
    return article.author.name;
}
```

In the following lesson, we’ll discuss a flag that is heavily related to `strictNullChecks`, the `strictPropertyInitialization` flag.