Type Checking an Interface with Custom User-Defined Type Guard

This lesson shows a way to type check by manually building an algorithm that aims to find unique characteristics in the structure.

WE'LL COVER THE FOLLOWING

- User-defined function
- User-defined function with the `in` keyword

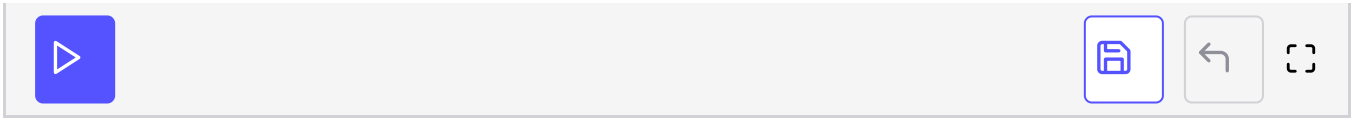
User-defined function

A custom user-defined type guard is a function that checks specific parts of the structure of an object to determine if it is one type or another. The advantage is that the type does not need to be altered with a branded property or use discriminator. However, it comes with the cost of foregoing functions that check every type.

The following code has a function at **line 21** which takes a single parameter of type `any`. The function checks for two members and because `any` is used can be anything. When the function returns `true`, the function will help TypeScript to narrow the type to the one defined after `is`. At **line 21**, the function narrows to `Person`.

```
interface Person {
  firstName: string;
  lastName: string;
}

interface Animal {
  name: string;
  numberOfLegs: number;
}

let p1: Person = {
  firstName: "First",
  lastName: "Last"
```

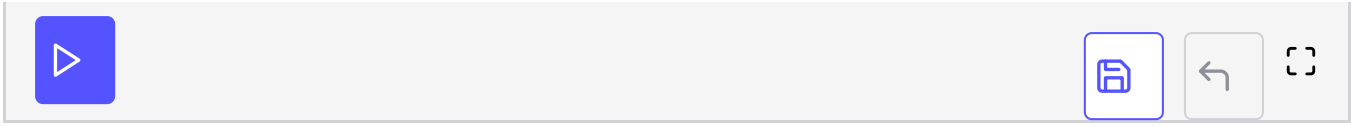
```
};

let p2: Animal = {
  name: "Tiger",
  numberOfLegs: 4
};

function isPerson(obj: any): obj is Person {
  return obj.firstName !== undefined && obj.lastName !== undefined;
}

console.log(isPerson(p1)); // A Person
console.log(isPerson(p2)); // Not a Person, does not have firstName and LastName
```



If the comparison is done in a union with exactly two different types, then the `else` statement is enough to allow TypeScript to narrow down to the alternative. In the following example, the closure of the `if` of **line 26** narrows the type to `Person` when the function `isPerson` returns `true`. The `else` will narrow to `Animal`.

```
interface Person {
  firstName: string;
  lastName: string;
}

interface Animal {
  name: string;
  numberOfLegs: number;
}

let p1: Person = {
  firstName: "First",
  lastName: "Last"
};

let p2: Animal = {
  name: "Tiger",
  numberOfLegs: 4
};

function isPerson(obj: any): obj is Person {
  return obj.firstName !== undefined && obj.lastName !== undefined;
}

function show(p: Person | Animal): void {
  if (isPerson(p)) {
    console.log(`It's a person with a first name: ${p.firstName}, and last: ${p.lastName}`)
  } else {
    console.log(`It's an animal: ${p.name}`)
  }
}

show(p1);
```

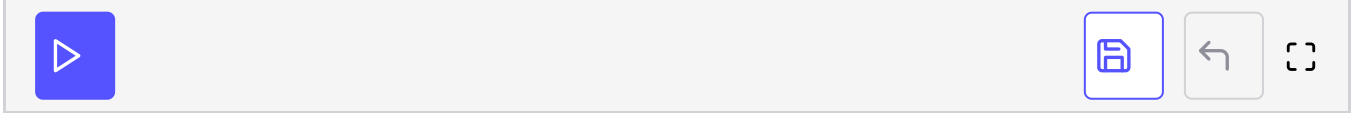
```
show(p2);
```



User-defined function with the `in` keyword

An alternative to checking of property is `undefined` is using the `in` operator which will return `true`, if present. The following code has a function `isPerson`. This time, the function uses `in` at **line 22** to verify if a member exists.

```
interface Person {
  firstName: string;
  lastName: string;
}

interface Animal {
  name: string;
  numberOfLegs: number;
}

let p1: Person = {
  firstName: "First",
  lastName: "Last"
};

let p2: Animal = {
  name: "Tiger",
  numberOfLegs: 4
};

function isPerson(obj: any): obj is Person {
  return "firstName" in obj;
}

function show(p: Person | Animal): void {
  if (isPerson(p)) {
    console.log(`It's a person with a first name: ${p.firstName}, and last: ${p.lastName}`)
  } else {
    console.log(`It's an animal: ${p.name}`)
  }
}

show(p1);
show(p2);
```



While having a function is a clean way to handle type, it is possible to skip

While having a function is a clean way to handle type, it is possible to skip straight to user-defined type guard functions and use `in` directly which

TypeScript will then narrow. The following code narrow properly when a union with **line 23** being a `Person` and in the `else`, at **line 25** an `Animal`.

```
interface Person {
  firstName: string;
  lastName: string;
}

interface Animal {
  name: string;
  numberOfLegs: number;
}

let p1: Person = {
  firstName: "First",
  lastName: "Last"
};

let p2: Animal = {
  name: "Tiger",
  numberOfLegs: 4
};

function show(p: Person | Animal): void {
  if ("firstName" in p) {
    console.log(`It's a person with a first name: ${p.firstName}, and last: ${p.lastName}`)
  } else {
    console.log(`It's an animal: ${p.name}`)
  }
}

show(p1);
show(p2);
```

The problem with using `in`, is as objects get complex, many comparisons of many properties are often necessary, causing the `if` statement to grow and then to be repeated along the code base. It is not recommended as a pattern to sprinkle conditional code, but it is supported by TypeScript.