Type Argument Constraints

This lesson talks about generic type argument constraints.

WE'LL COVER THE FOLLOWING

Overview

Sometimes it makes sense to put some restrictions on type arguments. Having these restrictions allows you to make assumptions about the type argument when referring to it inside some function.

You can express the restriction using the `extends` keyword. For example, you might decide to enforce the fact that `FormField` should only contain `string`, `number`, or `boolean` values.

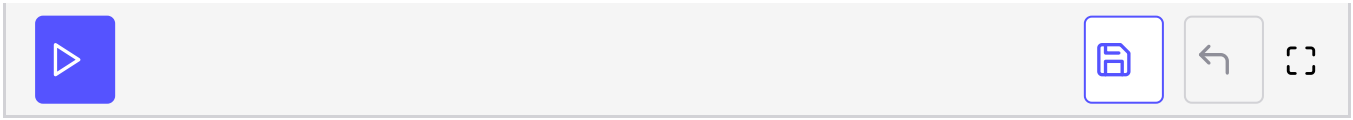
You will notice that after introducing the restriction, the `getFieldValue` function no longer compiles.

```
interface FormField<T extends string | number | boolean> {
    value?: T;
    defaultValue: T;
    isValid: boolean;
}

// Type 'T' does not satisfy the constraint 'string | number | boolean'
function getFieldValue<T>(field: FormField<T>): T { /* ... */ }
```

Run the code to see the error.

The reason for this error is that the `T` type argument of the function has no restrictions. You're trying to pass it to `FormField` which only accepts types that extend `string`, `number` or `boolean`. Therefore, you get a compile error. To get rid of the error, you need to put the same or stricter restrictions on the type argument `T`.

```
interface FormField<T extends string | number | boolean> {
    value?: T;
    defaultValue: T;
    isValid: boolean;
}

// Type 'T' does not satisfy the constraint 'string | number | boolean'
function getFieldValue<T extends string | number>(field: FormField<T>): T {
    return field.value ?? field.defaultValue;
}
```



Run the code to verify that there are no more errors.

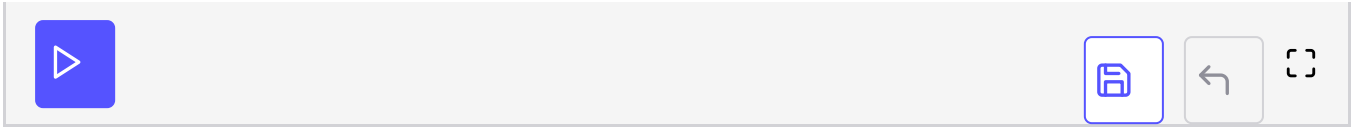
Constraining with index type query operator

A good example of using a type argument constraint is when typing a function that takes an object and a string representing the name of some property of this object. It's possible to type the function in such a way that if the provided string argument doesn't match any of the properties, the compiler will throw an error.

To achieve this, the function must take two type arguments: `T` and `K`. We can put a constraint on `K` and require it to extend `keyof T`. It's an example of using the *index type query operator* where `keyof K` returns a type that is a union of string literal types corresponding to the property names of `T`.

```
interface Person {
    name: string;
    age: number;
}

type PersonKeys = keyof Person; // 'name' | 'age'

function get<T, K extends keyof T>(object: T, key: K): T[K] {
    return object[key];
}

declare const person: Person;
get(person, 'name'); // ✓ No errors
```

```
get(person, 'name'); // ✅ No errors
get(person, 'foo'); // ⚪ Error!
```



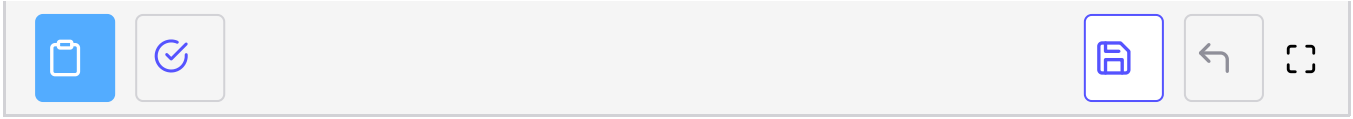
Run the code to see the error. Hover over `PersonKeys` to see the inferred type.

Exercise

Add types to the following `pick` function that takes an array of items and a property name and returns an array of values of the provided property of array elements.

*Note that the system can only verify that your code compiles without errors. To check your solution, click on **Show solution** and compare the code.*

```
function pick(array, key) {
    const results = [];
    for (let element of array) {
        results.push(element[key]);
    }
    return results;
}
```



The next lesson talks about an important theoretical topic that is closely related to generic types.