TypeScript Generic Constraints

Summary: in this tutorial, you'll learn about the generic constraints in TypeScript.

Introduction to generic constraints in TypeScript

Consider the following example:

```
function merge<U, V>(obj1: U, obj2: V) {
    return {
        ...obj1,
        ...obj2
    };
}
```

The `merge()` is a generic function that merges two objects. For example:

```
let person = merge(
    { name: 'John' },
    { age: 25 }
);

console.log(result);
```

Output:

```
{ name: 'John', age: 25 }
```

It works perfectly fine.

The `merge()` function expects two objects. However, it doesn't prevent you from passing a non-object like this:

```
let person = merge(
    { name: 'John' },
    25
);

console.log(person);
```

Output:

```
{ name: 'John' }
```

TypeScript doesn't issue any errors.

Instead of working with all types, you may want to add a constraint to the `merge()` function so that it works with objects only.

To do this, you need to list out the requirement as a constraint on what `U` and `V` types can be.

In order to denote the constraint, you use the `extends` keyword. For example:

```
function merge<U extends object, V extends object>(obj1: U, obj2: V) {
    return {
        ...obj1,
        ...obj2
    };
}
```

Because the `merge()` function is now constrained, it will no longer work with all types. Instead, it works with the `object` type only.

The following will result in an error:

```
let person = merge(
    { name: 'John' },
    25
);
```

Error:

```
Argument of type '25' is not assignable to parameter of type 'object'.
```

Using type parameters in generic constraints

TypeScript allows you to declare a type parameter constrained by another type parameter.

The following `prop()` function accepts an object and a property name. It returns the value of the property.

```
function prop<T, K>(obj: T, key: K) {
    return obj[key];
}
```

The compiler issues the following error:

```
Type 'K' cannot be used to index type 'T'.
```

To fix this error, you add a constraint to `K` to ensure that it is a key of `T` as follows:

```
function prop<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}
```

If you pass into the `prop` function a property name that exists on the `obj`, the compiler won't complain. For example:

```
let str = prop({ name: 'John' }, 'name');
console.log(str);
```

Output:

```
John
```

However, if you pass a key that doesn't exist on the first argument, the compiler will issue an error:

```
let str = prop({ name: 'John' }, 'age');
```

Error:

```
Argument of type '"age"' is not assignable to parameter of type '"name"'.
```

Summary

- Use `extends` keyword to constrain the type parameter to a specific type.
- Use `extends keyof` to constrain a type that is the property of another object.