Using some standard generic types

In this lesson, we start by understanding the syntax for generic types before consuming some of the standard generic types in TypeScript.

WE'LL COVER THE FOLLOWING

Understanding the generic type syntax

Generic type allows us to create a specific type by allowing us to pass types into it as parameters. The parameters are passed inside angle brackets. If there are multiple parameters, they are separated by a comma.

```
const myVar = GenericType<SpecificType1, SpecificType2, ...>
```

This will make more sense as we use some of the standard generic types below.

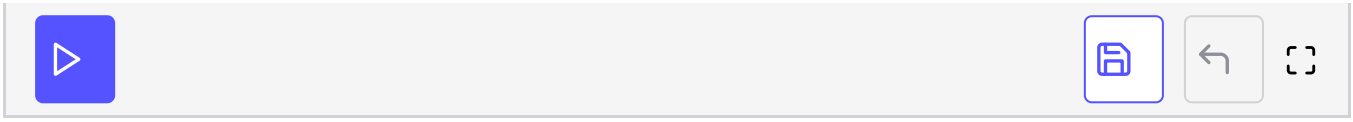
`Array<ItemType>`

One of the easiest generic types to understand is `Array`. In fact, we have used this before when we first created a strongly-typed array.

We pass the type we want the array elements to be into the `Array` type. Below is an example of an array of numbers.

</> TypeScript

```
let scores: Array<number>;
scores = [70, 65, 75];
```



Any type can be passed into generic parameters. So, if we have a `Coordinate` tuple type defined below:

```
type Coordinate = [number, number, number];
```

How can we use this `Coordinate` type to create an array of `Coordinate` type?

Show Answer

`Promise<ReturnType>`

We can use the `Promise` generic type to specify the return type of asynchronous code. We specify the type of the item that is eventually returned in the generic parameter.

In the example below, we use the `Promise` generic type to strongly-type what is returned from the `fetch` function:

</> TypeScript

```
const response: Promise<Response> = fetch("https://swapi.co/api/");
response.then(res => console.log(res.ok));
```

`Readonly<Type>`

The `Readonly` type simply adds the `readonly` keyword to each of the object properties that is passed into it.

So, if we have an `Action` type defined below:

</> TypeScript

```
type Action = {
  type: "fetchedName";

  data: string;
}
```

How can we use the `Readonly` generic type to make the `Action` type immutable?

Show Answer

`Partial<Type>`

The `Partial` type makes all the members of the type passed into it optional.

So, if we have the following type:

```
type Contact = {
  name: "Bob";
  email: "bob@someemail.com";
}
```

`Partial<Contact>` would be equivalent to the following type:

```
type Contact = {
  name?: "Bob";
  email?: "bob@someemail.com";
}
```

`Record<KeyType,ValueType>`

The `Record` type allows a key-value object type to be created. For example:

```
{
  rodj: {
    firstName: "Rod",
    surname: "James",
    score: 70
  },
  janes: {
    firstName: "Jane",
    surname: "Smith",
```

```
    score: 95
  },

  fredp: {
    firstName: "Fred",
    surname: "Peters",
    score: 60
  }
}
```

In the example above, `rodj`, `janes`, and `fredp` are the keys and the values are of type `{ firstName: string; surname: string; score: number; }`.

The `Record` type has two parameters for the types of the key and value. So, the above type can be represented as follows:

</> TypeScript

```
type Result = {
  firstName: string;
  surname: string;
  score: number;
}
type ResultRecord = Record<string, Result>;
```

How can we narrow the type of the record keys so that only `rodj`, `janes`, or `fredp` are accepted?

Show Answer

Wrap up

Generic types allow us to create types that are reusable in many different situations. This is because we can create a specific type for our use case from a generic type by passing our types into it as parameters.

Excellent, we are starting to understand generic types!

In the next lesson, we will begin to create our own generic types.