Creating generic interfaces

In this lesson, we will learn how to create generic interfaces.

WE'LL COVER THE FOLLOWING

Generic interface syntax

We can pass types into an interface that are used within its definition like we can to a function. The syntax for a generic interface is below:

```
interface InterfaceName<T1, T2, ...> {
    ...
}
```

The members of the interface can reference the generic types passed into it.

Generic interface example

A common use case for a generic interface is a generic form interface. This is because all forms have values, default values, validation rules, etc. but the specific fields differ from form to form.

Below is a simple generic form interface:

```
interface Form<T> {
  values: T;
}
```

We also have an interface for the fields on a contact form:

```
interface Contact {
  name: string;
  email: string;
}
```

How can we use both of these interfaces to create a strongly-typed version of the `contactForm` variable?

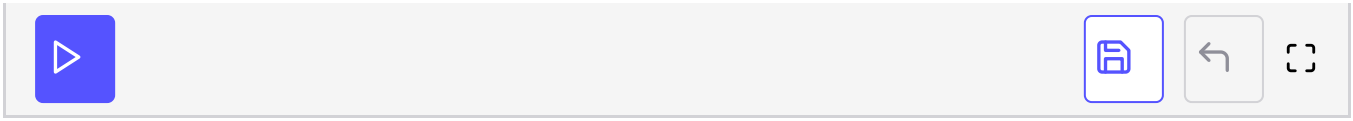
</> TypeScript

```
interface Form<T> {
  values: T;
}

interface Contact {
  name: string;
  email: string;
}

const contactForm = {
  values: {
    name: "Bob",
    email: "bob@someemail.com"
  }
}

console.log(contactForm);
```



Show Answer

We are now going to expand the `Form` interface to include a property for the form's validation errors. This will be based on the generic type passed into `Form`. Not all the fields will have validation errors. Let's add this `errors` property to our `Form` interface. Add the following code to the code widget above:

```
interface Form<T> {
  errors: {
    [P in keyof T]?: string;
  };
  values: T;
}
```

This is an advanced type that we haven't covered so far in this course. So, let's break this down:

- The type is in curly brackets, so we are constructing an object type.
- `[P in keyof T]` will put all the keys in the type `T` into a string literal union. This will be `"name" | "email"` for `contactForm`.
- `[P in keyof T]` is the property name of the object being constructed. So, for `contactForm`, the properties in the object are `name` and `email`.
- The `?` after the property name means the properties are optional.
- The type for the properties is `string`.
- So, for `contactForm`, the type for the errors is `{name?: string; email?: string}`.

Notice that a type error is raised on `errors` because it is a required property.

Add an empty errors object to our `contactForm` object.

```
const contactForm: Form<Contact> = {
 errors: {},
 values: { ... }
};
```

Notice that the type error disappears.

Add an error for the email to our `contactForm` object.

```
const contactForm: Form<Contact> = {
 errors: {
   email: "This must be a valid email address"
 },
 values: { ... }
};
```

What if we add an error for a field that doesn't belong to our `contactForm` object? Let's try this:

```
const contactForm: Form<Contact> = {
 errors: {
   age: "You must enter your age"
```

```
  },

  values: { ... }
};
```

Show Answer

Wrap up

Using generic interfaces allows generic types to be created that we can make specific by supplying our types as parameters.

More information can be found on generic interfaces in the TypeScript handbook.

In the next lesson, we will learn how to use generics with type aliases.