Creating interfaces

In this lesson, we learn what interfaces are and how to create them.

WE'LL COVER THE FOLLOWING

Understanding an interface

An interface allows a new type to be created with a name and structure. The structure includes all the properties and methods that the type has without any implementation.

Interfaces don't exist in JavaScript; they are only used by the TypeScript compiler type checking process.

Creating an interface

We create an interface with the `interface` keyword, followed by its name, followed by the properties and methods that make up the interface in curly brackets:

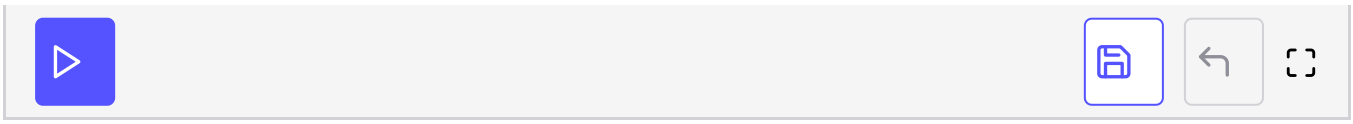
```
interface TypeName {
  propertyName: PropertyType;
  methodName: (paramName: ParamType) => MethodReturnType
}
```

As an exercise, create an interface called `ButtonProps` that has a `text` property of type `string` and an `onClick` method that has no parameters and doesn't return anything. We should be able to use this interface to create a `BuyButton` object beneath it. Run the code when you have finished to check that there are no errors.

</> TypeScript

```
// TODO - create the ButtonProps type using an interface

const BuyButton: ButtonProps = {
  text: "Buy",
  onClick: () => console.log("Buy")
}
```



Show Answer

Optional interface members

Interface members can be optional. Can you guess how you define an optional member?

Show Answer

Change the `ButtonProps` interface that we created earlier so that the `onClick` method is optional.

Show Answer

Readonly properties

A property can be readonly by putting a `readonly` keyword before the property name:

```
interface TypeName {
  readonly propertyName: PropertyType;
}
```

Change the `ButtonProps` interface that we are working on so that the `text` property is readonly.

Show Answer

What happens if we try to change the `text` property on the object after its declaration?

```
const BuyButton: ButtonProps = {
  text: "Buy",
  onClick: () => console.log("Buy")
}
BuyButton.text = "$20";  // is this okay?
```

Show Answer

Let's look at another code example below. Line 9 should generate a type error right? Run the code and find out.

</> TypeScript

```
interface Result {
  readonly name: string;
  readonly scores: number[];
}
let billScores: Result = {
  name: "Bill",
  scores: [90, 65, 80]
}
billScores.scores.push(70);
```

No type error is generated. Why do you think this is?

Show Answer

In this case we can put an additional `readonly` modifier before the array type as in the example below:

```
interface ImmutableResult {
  readonly name: string;
  readonly scores: readonly number[];
}
let tomScores: ImmutableResult = {
  name: "Tom",
  scores: [50, 95, 80]
}
tomScores.scores.push(70);
```

A type error will be generated on the last line now. Readonly arrays were introduced relatively recently in TypeScript 3.4.

Extending interfaces

Interfaces can *extend* other interfaces so that they inherit all the properties and methods from the interface being extended. We do this using the `extends` keyword after the new interface name and before the interface name that is being extended:

```
interface InterfaceA extends InterfaceB {
  ...
}
```

Create a new interface called `ColoredButtonProps` that extends the `ButtonProps` interface you created earlier in this lesson. Then add a `color` property of type `string`. We should be able to create the `GreenBuyButton` object beneath it.

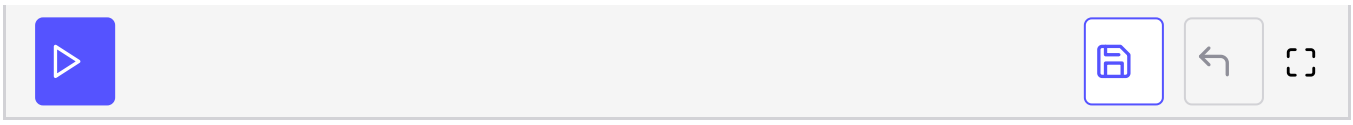
Don't forget to copy your `ButtonProps` code into the code widget below.

After you have done this, run the code to check that no errors are raised.

</> TypeScript

```
// TODO -  create a ColoredButtonProps interface that extends ButtonProps

const GreenBuyButton: ColoredButtonProps = {
  color: "Green",
  text: "Buy",
  onClick: () => console.log("Buy")
}
```



Show Answer

Interfaces vs type aliases

Interfaces create types like type aliases do, but interfaces seem more powerful. If this is the case, should we always use interfaces and forget about type aliases?

Historically, the capabilities of type aliases and interfaces were different, but now they are very similar. For example, type aliases can have optional and read-only properties, just like an interface. You can extend type aliases by using *intersection*, which we'll learn about later in this category of lessons.

So, it is generally personal preference as to which approach to use when creating types. Just be consistent with which method you use so that the code isn't confusing.

Wrap up

Interfaces are a powerful way of creating new TypeScript types that can be used throughout our code. The ability to extend existing interfaces helps us build objects from small lower-level interfaces.

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

In the next lesson, we will learn how to combine existing types to construct a new type.