Interfaces, Classes and More

Learn how to create interfaces and classes in this lesson.

Interfaces

In one of the prior examples I set the type of our variable like this: `Array<{label:string,value:string}>`

But what if the shape of our variable is much more complicated and we need to reuse it in multiple places? We can use an `interface` to define the shape that that variable should have.

```
interface Car {
  wheels: number;
  color: string;
  brand: string;
}
```

Be careful; an `interface` is not an `object`, so we use `;` and not `,`.

We can also set **optional properties** like this:

```
interface Car {
  wheels: number;
  color: string;
  brand: string;
  coupe?: boolean;
}
```

The `?` defines the property as optional, even if we create a new car object without it, `TypeScript` won't raise any error.

Maybe we don't want certain properties to be editable after the creation of the object, in that case we can mark them as **read only**.

```
interface Car {
  readonly wheels: number;
  color: string;
  brand: string;
  coupe?: boolean;
}
```

Upon creation of our car object, we'll be able to set the number of wheels but we won't be able to change it afterwards.

We can also create the shape for functions with interfaces, not just objects.

```
interface Greet {
  (greeting: string, name: string): string
}

let greetingFunction: Greet;

greetingFunction = (greeting: string, name: string): string => {
  console.log(`${greeting} ${name}`);
   return `${greeting} ${name}`;
}
greetingFunction('Bye', 'Alberto');
```

We create the shape that the function should have and then we assign the variable to a function of that said shape. If we assigned the variable to a function with a different shape, that would have thrown an error.

Extending Interfaces

An interface can extend another interface and inherit the members of the previous one:

```
interface Vehicle {
  wheels: number;

  color: string;
}

interface Airplane extends Vehicle {
  wings: number;
  rotors: number;
}
```

As you can see, an `object` of type airplane will expect to have all four properties, not just the two defined in the interface Airplane.

Classes

TypeScript classes are very similar to ES6 classes and allow us to perform prototypal inheritance to build reusable components for our applications.

Just as a reminder of how they look:

```
class Animal {
  eat = ()=> {
    console.log('gnam gnam')
  }
  sleep = () => {
    console.log('zzzz')
  }
}

class Human extends Animal {
  work = ()=> {
    console.log('zzzzzzz')
  }
}
const me = new Human();
me.work();
// zzzzzzz
me.eat();
// gnam gnam
me.sleep()
// zzzz
```

We have created two classes, the second one inherits two methods from the first one.

A difference with ES6 classes is that TypeScript allows us to define the way that class members will be accessed from our application.

If we want them to be accessible anywhere, we need to use the keyword `public`.

```
class Animal {
  public eat = ()=> {
    console.log('gnam gnam')
  }
  public sleep = () => {
    console.log('zzzz')
  }
}
const dog = new Animal();
dog.eat();
// gnam gnam
```

All class members are public in `JavaScript` so we cannot restrict access to them.

`TypeScript` also offers us to mark a member as `private` which means that it will not be accessible from outside of the class.

```
class Animal {
  public eat = ()=> {
    console.log('gnam gnam');
  }
  public sleep = () => {
    console.log('zzzz')
  }
}

class Human extends Animal {
  private work = ()=> {
    console.log('zzzzzzz');
  }
}

const me = new Human();
me.work();
// Property 'work' is private and only accessible within class 'Human'
```

We can also mark a member as `protected` so that it will be accessible only

```
class Human {
  protected work: ()=> {
    console.log('zzzzzzz')
  }
}

class Adult extends Human {
  public doWork = () => {
    console.log(this.work)
  }
}
```

The class `Adult` extends the class `Human`, therefore it can access the same `protected` methods.

Intersection Types and Union Types

We’ve seen how we can define basic types with `TypeScript`, now let’s dive into more advanced types.

Intersection Types

As the name implies, intersection types allow us to join together multiple types. Let’s look at how to use them:

```
interface Person {
  sex: 'male' | 'female' | 'N/A',
  age:number,
}

interface WorkerAdult {
  job: string
}

type Adult =  Person & WorkerAdult;

const me: Adult = {
  sex: 'male',
  age: 27,
  job: 'software developer'
}
console.log(me);
// { sex: 'male', age: 27, job: 'software developer' }
```

As you can see, we have two distinct types: `Person` and `Worker`. We combined them together to create the type `Adult` as a combination of the two.

Be careful when combining types that have two of the same properties but of different types, since that will cause an error in the compiler.

Union Types

```
const attendee = string | string[];
```

The variable can be either a `string` or an `array of strings`.

Another example may be:

```
const identifier = string | number | string[];
```

We use the pipe (`|`) symbol to separate each type.

Remember that we can only access common members of all types of the union.

```
interface Kid {
  age:number
}
interface Adult {
  age: number,
  job: string
}

function person(): Kid | Adult {
  return { age: 27 }
  }

const me = person();
me.age // ok
me.job // error
```

The property `job` exists only on the interface `Adult` and not on the `Kid` one, therefore we can’t access it in our union type.

Let’s revise with a Quiz.