100 XP

# Design considerations

5 minutes

TypeScript offers a couple of key ways to define the structure of objects - classes and interfaces. You may be wondering when it's best to use each.

## When to use interfaces

Interfaces are a TypeScript design-time construct. Because JavaScript does not have a concept of interfaces, they are removed when TypeScript is transpiled to JavaScript. This means they are completely weightless, take up no space in the resulting file, and have no negative impact on the code that will be executed.

Unlike other programming languages where interfaces can only be used with classes, TypeScript allows you to use an interface to define a data structure without the need for a class. You can use interfaces to define parameter objects for functions, define the structure for various framework properties, and define how objects look from remote services or APIs.

If you were creating a full-stack application with both client and server implementations, you will typically need to define how data will be structured. If the data in question was to store information about a dog, for example, you might create an interface that looks like this:

TypeScript

```
interface Dog {
    id?: number;
    name: string;
    age: number;
    description: string;
}
```

This interface could be used in a shared module for both your client and server code, ensuring the data structure is the same on both sides. On the client, you might have code to retrieve the dog from the server API you define, which looks like the following:

TypeScript

```
async loadDog(id: number): Dog {
    return await (await fetch('demoUrl')).json() as Dog;
}
```

By using the interface, `loadDog` will let TypeScript know the structure of the object. You don't need to create a class to ensure this works.

# When to use classes

The key difference between interfaces and classes in any programming language is that classes allow you to define implementation details. Interfaces solely define how data is structured. Classes allow you to define methods, fields, and properties. Classes also provide a way to template objects, defining default values.

Returning to the example above, on the server you may want to add code to load or save a dog to the database. A common technique for managing data in a database is to use what's known as the "active record pattern", meaning the object itself has `save`, `load` and similar methods. We can use the `Dog` interface defined above to ensure we have the same properties and structure, while adding the necessary code to perform the operations.

TypeScript

```
class DogRecord implements Dog {
    id: number;
    name: string;
    age: number;
    description: string;

    constructor({name, age, description, id = 0}: Dog) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.description = description;
    }

    static load(id: number): DogRecord {
        // code to load dog from database
        return dog;
    }

    save() {
        // code to save dog to database
```

```
    }
}
```

# Summary

As you continue to use TypeScript you will find many new instances where interfaces, in particular, will make your life as a developer easier. One key feature of TypeScript to remember about interfaces is they **do not** require a class. This allows you to use them whenever you need the ability to define a data structure without having to create a full class implementation.

## Next unit: Lab - Convert three TypeScript functions to a class definition

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆