TypeScript Objects

Objects of TypeScript

Motivation

With a growing understanding of typescript, we will now look deeper into objects.

Objects

We have already seen objects being used in TypeScript and its specialities.

Also have we seen several forms, simple:

```
1  function greet(person: { name: string; age: number }) {
2   return "Hello " + person.age;
3 }
```

Objects

With an interface:

```
interface Person {
  name: string;
  age: number;
}

function greet(person: Person) {
  return "Hello " + person.age;
}
```

Objects

With a type alias:

```
type Person = {
  name: string;
  age: number;
};

function greet(person: Person) {
  return "Hello " + person.age;
}
```

Objects

But sometimes we need to have optional properties.

Let us look at the following pseudocode example.

Objects

Shapes appear in different kinds, and therefore we need a way to have optional properties.

Objects

Shapes appear in different kinds, and therefore we need a way to have optional properties.

```
interface PaintOptions {
   shape: Shape;
   xPos?: number;
   yPos?: number;
}
 function paintShape(opts: PaintOptions) {
   // ...
}
 const shape = getShape();
 paintShape({ shape });
 paintShape({ shape, xPos: 100 });
 paintShape({ shape, yPos: 100 });
 paintShape({ shape, xPos: 100, yPos: 100 });
```

Objects

Shapes appear in different kinds, and therefore we need a way to have optional properties.

```
interface PaintOptions {
   shape: Shape;
   xPos?: number;
   yPos?: number;
}
 function paintShape(opts: PaintOptions) {
   // ...
}
 const shape = getShape();
 paintShape({ shape });
 paintShape({ shape, xPos: 100 });
 paintShape({ shape, yPos: 100 });
 paintShape({ shape, xPos: 100, yPos: 100 });
```

As you can see, `shape` is needed but **`xPos`** and **`yPos`** can be omitted.

Objects

In other circumstances you want to make sure that your property is read-only:

```
1  interface SomeType {
2    readonly prop: string;
3  }
4
5  function doSomething(obj: SomeType) {
6    // We can read from 'obj.prop'.
7    console.log(`prop has the value '${obj.prop}'.`);
8    // But we can't re-assign it.
9    obj.prop = "hello";
10   // Cannot assign to 'prop' because it is a read-only property.
11 }
```

Using the readonly modifier does not necessarily imply that a value is **`totally immutable`**. It just means the property itself can not be rewritten.

Objects

In other circumstances you want to make sure that your property is read-only:

```
1  interface SomeType {
2   readonly prop: string;
3  }
4
5  function doSomething(obj: SomeType) {
6    // We can read from 'obj.prop'.
7    console.log(`prop has the value '${obj.prop}'.`);
8    // But we can't re-assign it.
9    obj.prop = "hello";
10   // Cannot assign to 'prop' because it is a read-only property.
11 }
```

Using the readonly modifier does not necessarily imply that a value is **`totally immutable`**. It just means the property itself can not be rewritten.

Objects

In other circumstances you want to make sure that your property is read-only:

```
 1  interface SomeType {
 2   readonly prop: string;
 3 }
 4
 5 function doSomething(obj: SomeType) {
 6   // We can read from 'obj.prop'.
 7   console.log(`prop has the value '${obj.prop}'.`);
 8   // But we can't re-assign it.
 9   obj.prop = "hello";
10   // Cannot assign to 'prop' because it is a read-only property.
11 }
```

Using the readonly modifier does not necessarily imply that a value is **`totally immutable`**. It just means the property itself can not be rewritten.

Objects

Another quite common example is that you have two or more slightly different interfaces. If this is the case, you probably face an interface that needs to be extended.

Have a look at the following interfaces:

```typescript
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  postalCode: string;
}

interface AddressWithUnit {
  name?: string;
  unit: string;
  street: string;
  city: string;
  postalCode: string;
}
```

The only difference is the added **`unit`** in **`AddressWithUnit`**.

Objects

If OOP is not new to you, you might immediately think you can extend interfaces.

Luckily, this is the case; please see the improved structure.

Objects

If OOP is not new to you, you might immediately think you can extend interfaces.

Luckily, this is the case; please see the improved structure.

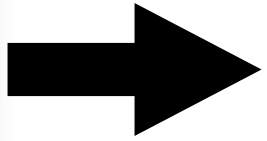
```
1  interface BasicAddress {
2    name?: string;
3    street: string;
4    city: string;
5    postalCode: string;
6  }
7
8  interface AddressWithUnit {
9    name?: string;
10   unit: string;
11   street: string;
12   city: string;
13   postalCode: string;
14 }
```

Objects

If OOP is not new to you, you might immediately think you can extend interfaces.

Luckily, this is the case; please see the improved structure.

```
1  interface BasicAddress {
2    name?: string;
3    street: string;
4    city: string;
5    postalCode: string;
6  }
7
8  interface AddressWithUnit {
9    name?: string;
10   unit: string;
11   street: string;
12   city: string;
13   postalCode: string;
14 }
```



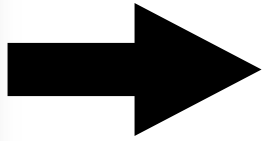
Objects

If OOP is not new to you, you might immediately think you can extend interfaces.

Luckily, this is the case; please see the improved structure.

```typescript
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  postalCode: string;
}

interface AddressWithUnit {
  name?: string;
  unit: string;
  street: string;
  city: string;
  postalCode: string;
}
```



```typescript
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}

interface AddressWithUnit extends BasicAddress {
  unit: string;
}
```

Objects

Multiple interface extends are also possible:

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
}

interface ColorfulCircle extends Colorful, Circle {}
const cc: ColorfulCircle = {
  color: "red",
  radius: 42,
};
```

Objects intersection Types

Interfaces allow us to build up new types from other types by extending them.

TypeScript provides another construct mainly used to combine existing object types. Consider the following:

```
1 interface Colorful {
2 color: string;
3 }
4
5 interface Circle {
6 radius: number;
7 }
8
9 type ColorfulCircle = Colorful & Circle;
```

Objects Exercise

Please think about a structure of at least three levels,

For example, Animal -> Pet -> Dog and try to implement extending useful interfaces.

Also, try to do this with intersection and extends in *04_objects/exercises/01_objects.ts*.

Objects Solution

```
interface Animal {
    name: string
}

interface Pet extends Animal {
    tame: boolean
}

interface Dog extends Animal {
    color: string,
    age: number
}


function printMyDog(dog: Dog) {
    console.log(`My dog is called: '${dog.name}'.`);
}


printMyDog({name: "flipper", age: 5, color: "beige"})

```

Objects Solution

```
interface Animal {
    name: string
}

interface Pet extends Animal {
    tame: boolean
}

interface Dog extends Animal {
    color: string,
    age: number
}


function printMyDog(dog: Dog) {
    console.log(`My dog is called: '${dog.name}'.`);
}


printMyDog({name: "flipper", age: 5, color: "beige"})

```

Objects Solution

```
interface Animal {
    name: string
}

interface Pet extends Animal {
    tame: boolean
}

interface Dog extends Animal {
    color: string,
    age: number
}


function printMyDog(dog: Dog) {
    console.log(`My dog is called: '${dog.name}'.`);
}


printMyDog({name: "flipper", age: 5, color: "beige"})

```

Objects Solution

```
interface Animal {
    name: string
}

interface Pet extends Animal {
    tame: boolean
}

interface Dog extends Animal {
    color: string,
    age: number
}


function printMyDog(dog: Dog) {
    console.log(`My dog is called: '${dog.name}'.`);
}


printMyDog({name: "flipper", age: 5, color: "beige"})

```

Objects Solution

```
interface Animal {
    name: string
}

interface Pet extends Animal {
    tame: boolean
}

interface Dog extends Animal {
    color: string,
    age: number
}


function printMyDog(dog: Dog) {
    console.log(`My dog is called: '${dog.name}'.`);
}


printMyDog({name: "flipper", age: 5, color: "beige"})

```

Objects Solution

```
interface Animal {
    name: string
}

interface Pet extends Animal {
    tame: boolean
}

interface Dog extends Animal {
    color: string,
    age: number
}


function printMyDog(dog: Dog) {
    console.log(`My dog is called: '${dog.name}'.`);
}


printMyDog({name: "flipper", age: 5, color: "beige"})

```

Good Job!

Objects are getting clearer now, let us cover other topics.

End

That was all for this chapter