

TypeScript 3.2 is now available. Download ([/#download-links](#)) our latest version today!

Documentation ▼

Interfaces

Introduction

One of TypeScript's core principles is that type-checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The type-checker checks the call to `printLabel`. The `printLabel` function has a single parameter that requires that the object passed in has a property called `label` of type `string`. Notice that our object actually has more properties than this, but the compiler only checks that *at least* the ones required are present and match the types required. There are some cases where TypeScript isn't as lenient, which we'll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the `label` property that is a string:

```
interface LabelledValue {
    label: string;
}

function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The interface `LabelledValue` is a name we can now use to describe the requirement in the previous example. It still represents having a single property called `label` that is of type `string`. Notice we didn't have to explicitly say that the object we pass to `printLabel` implements this interface like we might have to in other languages. Here, it's only the shape that matters. If the object we pass to the function meets the requirements listed, then it's allowed.

It's worth pointing out that the type-checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like “option bags” where you pass an object to a function that only has a couple of properties filled in.

Here’s an example of this pattern:

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
  let newSquare = {color: "white", area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a `?` at the end of the property name in the declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the `color` property in `createSquare`, we would get an error message letting us know:

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = {color: "white", area: 100};
  if (config.clor) {
    // Error: Property 'clor' does not exist on type 'SquareConfig'
    newSquare.color = config.clor;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: "black"});
```

Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by putting `readonly` before the name of the property:

```
interface Point {  
    readonly x: number;  
    readonly y: number;  
}
```

You can construct a `Point` by assigning an object literal. After the assignment, `x` and `y` can't be changed.

```
let p1: Point = { x: 10, y: 20 };  
p1.x = 5; // error!
```

TypeScript comes with a `ReadonlyArray<T>` type that is the same as `Array<T>` with all mutating methods removed, so you can make sure you don't change your arrays after creation:

```
let a: number[] = [1, 2, 3, 4];  
let ro: ReadonlyArray<number> = a;  
ro[0] = 12; // error!  
ro.push(5); // error!  
ro.length = 100; // error!  
a = ro; // error!
```

On the last line of the snippet you can see that even assigning the entire `ReadonlyArray` back to a normal array is illegal. You can still override it with a type assertion, though:

```
a = ro as number[];
```

readonly vs const

The easiest way to remember whether to use `readonly` or `const` is to ask whether you're using it on a variable or a property. Variables use `const` whereas properties use `readonly`.

Excess Property Checks

In our first example using interfaces, TypeScript lets us pass `{ size: number; label: string; }` to something that only expected a `{ label: string; }`. We also just learned about optional properties, and how they're useful when describing so-called "option bags".

However, combining the two naively would let you to shoot yourself in the foot the same way you might in JavaScript. For example, taking our last example using `createSquare`:

```
interface SquareConfig {  
    color?: string;  
    width?: number;  
}  
  
function createSquare(config: SquareConfig): { color: string; area: number } {  
    // ...  
}  
  
let mySquare = createSquare({ colour: "red", width: 100 });
```

Notice the given argument to `createSquare` is spelled *colour* instead of `color`. In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the `width` properties are compatible, there's no `color` property present, and the extra `colour` property is insignificant.

However, TypeScript takes the stance that there's probably a bug in this code. Object literals get special treatment and undergo *excess property checking* when assigning them to other variables, or passing them as arguments. If an object literal has any properties that the "target type" doesn't have, you'll get an error.

```
// error: 'colour' not expected in type 'SquareConfig'
let mySquare = createSquare({ colour: "red", width: 100 });
```

Getting around these checks is actually really simple. The easiest method is to just use a type assertion:

```
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

However, a better approach might be to add a string index signature if you're sure that the object can have some extra properties that are used in some special way. If `SquareConfig` can have `color` and `width` properties with the above types, but could *also* have any number of other properties, then we could define it like so:

```
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}
```

We'll discuss index signatures in a bit, but here we're saying a `SquareConfig` can have any number of properties, and as long as they aren't `color` or `width`, their types don't matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since `squareOptions` won't undergo excess property checks, the compiler won't give you an error.

```
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

Keep in mind that for simple code like above, you probably shouldn't be trying to "get around" these checks. For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs. That means if you're running into excess property checking problems for something like option bags, you might need to revise some of your type declarations. In this instance, if it's okay to pass an object with both a `color` or `colour` property to `createSquare`, you should fix up the definition of `SquareConfig` to reflect that.

Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and type.

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    return result > -1;
}
```

For function types to correctly type-check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
    let result = src.search(sub);
    return result > -1;
}
```

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not want to specify types at all, TypeScript's contextual typing can infer the argument types since the function value is assigned directly to a variable of type `SearchFunc`. Here, also, the return type of our function expression is implied by the values it returns (here `false` and `true`). Had the function expression returned numbers or strings, the type-checker would have warned us that return type doesn't match the return type described in the `SearchFunc` interface.

```
let mySearch: SearchFunc;
mySearch = function(src, sub) {
    let result = src.search(sub);
    return result > -1;
}
```

Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can "index into" like `a[10]`, or `ageMap["daniel"]`. Indexable types have an *index signature* that describes the types we can use to index into the object, along with the corresponding return types when indexing. Let's take an example:

```
interface StringArray {
    [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

Above, we have a `StringArray` interface that has an index signature. This index signature states that when a `StringArray` is indexed with a `number`, it will return a `string`.

There are two types of supported index signatures: `string` and `number`. It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a `number`, JavaScript will actually convert that to a `string` before indexing into an object. That means that indexing with `100` (a `number`) is the same thing as indexing with `"100"` (a `string`), so the two need to be consistent.

```

class Animal {
    name: string;
}
class Dog extends Animal {
    breed: string;
}

// Error: indexing with a numeric string might get you a completely separate type of Animal!
interface NotOkay {
    [x: number]: Animal;
    [x: string]: Dog;
}

```

While string index signatures are a powerful way to describe the “dictionary” pattern, they also enforce that all properties match their return type. This is because a string index declares that `obj.property` is also available as `obj["property"]`. In the following example, `name`’s type does not match the string index’s type, and the type-checker gives an error:

```

interface NumberDictionary {
    [index: string]: number;
    length: number;    // ok, length is a number
    name: string;       // error, the type of 'name' is not a subtype of the indexer
}

```

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

```

interface ReadonlyStringArray {
    readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!

```

You can’t set `myArray[2]` because the index signature is readonly.

Class Types

Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

```

interface ClockInterface {
    currentTime: Date;
}

class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}

```

You can also describe methods in an interface that are implemented in the class, as we do with `setTime` in the below example:

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

Difference between the static and instance sides of classes

When working with classes and interfaces, it helps to keep in mind that a class has *two* types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

```
interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, `ClockConstructor` for the constructor and `ClockInterface` for the instance methods. Then for convenience we define a constructor function `createClock` that creates instances of the type that is passed to it.

```
interface ClockConstructor {
    new (hour: number, minute: number): ClockInterface;
}
interface ClockInterface {
    tick();
}

function createClock(ctor: ClockConstructor, hour: number, minute: number): ClockInterface {
    return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("beep beep");
    }
}
class AnalogClock implements ClockInterface {
    constructor(h: number, m: number) { }
    tick() {
        console.log("tick tock");
    }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);
```

Because `createClock`'s first parameter is of type `ClockConstructor`, in `createClock(AnalogClock, 7, 32)`, it checks that `AnalogClock` has the correct constructor signature.

Extending Interfaces

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

An interface can extend multiple interfaces, creating a combination of all of the interfaces.


```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript's dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully describe the shape of the type.

Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

This is useful when you have a large inheritance hierarchy, but want to specify that your code works with only subclasses that have certain properties. The subclasses don't have to be related besides inheriting from the base class. For example:

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control implements SelectableControl {
    select() { }
}

class TextBox extends Control {
    select() { }
}


// Error: Property 'state' is missing in type 'Image'.
class Image implements SelectableControl {
    select() { }
}

class Location {
}
```

In the above example, `SelectableControl` contains all of the members of `Control`, including the private `state` property. Since `state` is a private member it is only possible for descendants of `Control` to implement `SelectableControl`. This is because only descendants of `Control` will have a `state` private member that originates in the same declaration, which is a requirement for private members to be compatible.

Within the `Control` class it is possible to access the `state` private member through an instance of `SelectableControl`. Effectively, a `SelectableControl` acts like a `Control` that is known to have a `select` method. The `Button` and `TextBox` classes are subtypes of `SelectableControl` (because they both inherit from `Control` and have a `select` method), but the `Image` and `Location` classes are not.

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft