Skip to main content

Documentation

Declaration Merging

Introduction

Some of the unique concepts in TypeScript describe the shape of JavaScript objects at the type level. One example that is especially unique to TypeScript is the concept of 'declaration merging'. Understanding this concept will give you an advantage when working with existing JavaScript. It also opens the door to more advanced abstraction concepts.

For the purposes of this article, "declaration merging" means that the compiler merges two separate declarations declared with the same name into a single definition. This merged definition has the features of both of the original declarations. Any number of declarations can be merged; it's not limited to just two declarations.

Basic Concepts

In TypeScript, a declaration creates entities in at least one of three groups: namespace, type, or value. Namespace-creating declarations create a namespace, which contains names that are accessed using a dotted notation. Type-creating declarations do just that: they create a type that is visible with the declared shape and bound to the given name. Lastly, value-creating declarations create values that are visible in the output JavaScript.

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

Understanding what is created with each declaration will help you understand what is merged when you perform a declaration merge.

Merging Interfaces

The simplest, and perhaps most common, type of declaration merging is interface merging. At the most basic level, the merge mechanically joins the members of both declarations into a single interface with the same name.

```
interface Box {
    height: number;
    width: number;
}

interface Box {
    scale: number;
}

let box: Box = {height: 5, width: 6, scale: 10};
```

Non-function members of the interfaces should be unique. If they are not unique, they must be of the same type. The compiler will issue an error if the interfaces both declare a non-function member of the same name, but of different types.

For function members, each function member of the same name is treated as describing an overload of the same function. Of note, too, is that in the case of interface `A` merging with later interface `A`, the second interface will have a higher precedence than the first.

That is, in the example:

```
interface Cloner {
    clone(animal: Animal): Animal;
}

interface Cloner {
    clone(animal: Sheep): Sheep;
}

interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
}
```

The three interfaces will merge to create a single declaration as so:

```
interface Cloner {
    clone(animal: Dog): Dog;
    clone(animal: Cat): Cat;
    clone(animal: Sheep): Sheep;
    clone(animal: Animal): Animal;
}
```

Notice that the elements of each group maintains the same order, but the groups themselves are merged with later overload sets ordered first.

One exception to this rule is specialized signatures. If a signature has a parameter whose type is a *single* string literal type (e.g. not a union of string literals), then it will be bubbled toward the top of its merged overload list.

For instance, the following interfaces will merge together:

```
interface Document {
    createElement(tagName: any): Element;
}
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
}
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

The resulting merged declaration of `Document` will be the following:

```
interface Document {
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: string): HTMLElement;
    createElement(tagName: any): Element;
}
```

Merging Namespaces

Similarly to interfaces, namespaces of the same name will also merge their members. Since namespaces create both a namespace and a value, we need to understand how both merge.

To merge the namespaces, type definitions from exported interfaces declared in each namespace are themselves merged, forming a single namespace with merged interface definitions inside.

To merge the namespace value, at each declaration site, if a namespace already exists with the given name, it is further extended by taking the existing namespace and adding the exported members of the second namespace to the first.

The declaration merge of `Animals` in this example:

```
namespace Animals {
    export class Zebra { }
}

namespace Animals {
    export interface Legged { numberOfLegs: number; }
    export class Dog { }
}
```

is equivalent to:

```
namespace Animals {
    export interface Legged { numberOfLegs: number; }

    export class Zebra { }
    export class Dog { }
}
```

This model of namespace merging is a helpful starting place, but we also need to understand what happens with non-exported members. Non-exported members are only visible in the original (un-merged) namespace. This means that after merging, merged members that came from other declarations cannot see non-exported members.

We can see this more clearly in this example:

```
namespace Animal {
    let haveMuscles = true;

    export function animalsHaveMuscles() {
        return haveMuscles;
    }
}

namespace Animal {
    export function doAnimalsHaveMuscles() {
        return haveMuscles;  // Error, because haveMuscles is not accessible here
    }
}
```

Because `haveMuscles` is not exported, only the `animalsHaveMuscles` function that shares the same un-merged namespace can see the symbol. The `doAnimalsHaveMuscles` function, even though it's part of the merged `Animal` namespace can not see this un-exported member.

Merging Namespaces with Classes, Functions, and Enums

Namespaces are flexible enough to also merge with other types of declarations. To do so, the namespace declaration must follow the declaration it will merge with. The resulting declaration has properties of both declaration types. TypeScript uses this capability to model some of the patterns in JavaScript as well as other programming languages.

Merging Namespaces with Classes

This gives the user a way of describing inner classes.

```
class Album {
    label: Album.AlbumLabel;
}
namespace Album {
    export class AlbumLabel { }
}
```

The visibility rules for merged members is the same as described in the 'Merging Namespaces' section, so we must export the `AlbumLabel` class for the merged class to see it. The end result is a class managed inside of another class. You can also use namespaces to add more static members to an existing class.

In addition to the pattern of inner classes, you may also be familiar with JavaScript practice of creating a function and then extending the function further by adding properties onto the function. TypeScript uses declaration merging to build up definitions like this in a type-safe way.

```
function buildLabel(name: string): string {
    return buildLabel.prefix + name + buildLabel.suffix;
}

namespace buildLabel {
    export let suffix = "";
    export let prefix = "Hello, ";
}

console.log(buildLabel("Sam Smith"));
```

Similarly, namespaces can be used to extend enums with static members:

```
enum Color {
    red = 1,
    green = 2,
    blue = 4
}

namespace Color {
    export function mixColor(colorName: string) {
        if (colorName == "yellow") {
            return Color.red + Color.green;
        }
        else if (colorName == "white") {
            return Color.red + Color.green + Color.blue;
        }
        else if (colorName == "magenta") {
            return Color.red + Color.blue;
        }
        else if (colorName == "cyan") {
            return Color.green + Color.blue;
        }
    }
}
```

Disallowed Merges

Not all merges are allowed in TypeScript. Currently, classes can not merge with other classes or with variables. For information on mimicking class merging, see the Mixins in TypeScript (./mixins.html) section.

Module Augmentation

Although JavaScript modules do not support merging, you can patch existing objects by importing and then updating them. Let's look at a toy Observable example:

```
// observable.js
export class Observable<T> {
    // ... implementation left as an exercise for the reader ...
}

// map.js
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}
```

This works fine in TypeScript too, but the compiler doesn't know about `Observable.prototype.map`. You can use module augmentation to tell the compiler about it:

```
// observable.ts stays the same
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
    interface Observable<T> {
        map<U>(f: (x: T) => U): Observable<U>;
    }
}
Observable.prototype.map = function (f) {
    // ... another exercise for the reader
}


// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map(x => x.toFixed());
```

The module name is resolved the same way as module specifiers in `import` / `export`. See Modules (./modules.html) for more information. Then the declarations in an augmentation are merged as if they were declared in the same file as the original. However, you can't declare new top-level declarations in the augmentation – just patches to existing declarations.

Global augmentation

You can also add declarations to the global scope from inside a module:

```
// observable.ts
export class Observable<T> {
    // ... still no implementation ...
}

declare global {
    interface Array<T> {
        toObservable(): Observable<T>;
    }
}

Array.prototype.toObservable = function () {
    // ...
}
```

Global augmentations have the same behavior and limits as module augmentations.