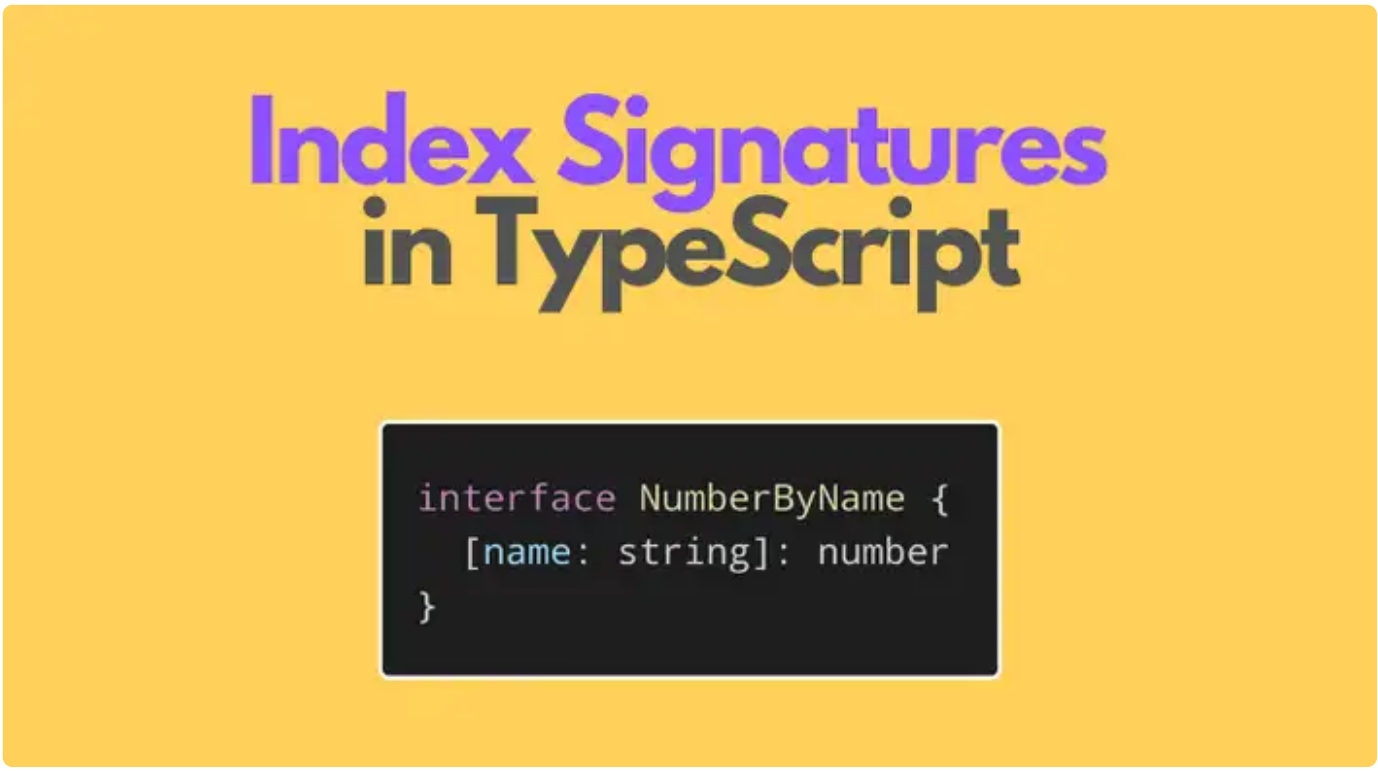

Index Signatures in TypeScript

Updated March 21, 2023

typescript object 19 Comments

You have 2 objects that describe the salary of 2 software developers:

```
const salary1 = {
  baseSalary: 100_000,
  yearlyBonus: 20_000
};

const salary2 = {
  contractSalary: 110_000
};
```

You want to implement a function that returns the total remuneration based on the salary object:

```
function totalSalary(salaryObject: ???) {
  let total = 0;
  for (const name in salaryObject) {
    total += salaryObject[name];
  }
  return total;
}

console.log(totalSalary(salary1)); // => 120_000
console.log(totalSalary(salary2)); // => 110_000
```

How would you annotate `salaryObject` parameter of `totalSalary()` function to accept objects with key as string and value as number?

The answer is to use an index signature!

Let's find what are TypeScript index signatures and when they're needed.

Table of Contents

1. Why index signature

The idea of the index signatures is to type objects of unknown structure when you only know the key and value types.

An index signature fits the case of the salary parameter: the function should accept salary objects of different structures — just make sure that object values are numbers.

Let's annotate the `salaryObject` parameter with an index signature:

```
function totalSalary(salaryObject: { [key: string]: number }) {
  let total = 0;
  for (const name in salaryObject) {
    total += salaryObject[name];
  }
  return total;
}

console.log(totalSalary(salary1)); // => 120_000
console.log(totalSalary(salary2)); // => 110_000
```

`{ [key: string]: number }` is the index signature, which tells TypeScript that `salaryObject` has to be an object with `string` type as key and `number` type as value.

Now the `totalSalary()` accepts as arguments both `salary1` and `salary2` objects, since they are objects with number values.

However, the function would not accept an object that has, for example, strings as values:

```
const salary3 = {
  baseSalary: '100 thousands'
};

// Type error:
// Argument of type '{ baseSalary: string; }' is not assignable to par
//   Property 'baseSalary' is incompatible with index signature.
//     Type 'string' is not assignable to type 'number'.
totalSalary(salary3);
```

2. Index signature syntax

The syntax of an index signature is simple and looks similar to the syntax of a property. But with one difference: write the type of the key inside the square brackets: `{ [key: KeyType]: ValueType }`.

Here are a few examples of index signatures.

`string` type is the key and value:

```
interface StringByString {
  [key: string]: string;
}

const heroesInBooks: StringByString = {
  'Gunslinger': 'The Dark Tower',
  'Jack Torrance': 'The Shining'
};
```

The `string` type is the key, the value can be a `string`, `number`, or `boolean`:

```
interface Options {
  [key: string]: string | number | boolean;
  timeout: number;
}

const options: Options = {
  timeout: 1000,
  timeoutMessage: 'The request timed out!',
  isFileUpload: false
};
```

`Options` interface also has a field `timeout`, which works fine near the index signature.

The key of the index signature can only be a `string`, `number`, or `symbol`. Other types are not allowed:

```
interface OopsDictionary {
  // Type error:
  // An index signature parameter type must be 'string', 'number',
  // 'symbol', or a template literal type.
```

```
  [key: boolean]: string;
}
```

3. Index signature caveats

The index signatures in TypeScript have a few caveats you should be aware of.

3.1 Non-existing properties

What would happen if you try to access a non-existing property of an object whose index signature is `{ [key: string]: string }`?

As expected, TypeScript infers the type of the value to `string`. But if you check the runtime value — it's `undefined`:

```
interface StringByString {
  [key: string]: string;
}

const object: StringByString = {};

const value = object['nonExistingProp'];
console.log(value); // => undefined
```

`value` variable is a `string` type according to TypeScript, however, its runtime value is `undefined`.

The index signature maps a key type to a value type — that's all. If you don't make that mapping correct, the value type can deviate from the actual runtime data type.

To make typing more accurate, mark the indexed value as `string` or `undefined`. Doing so, TypeScript becomes aware that the properties you access might not exist:

```
interface StringByString {
  [key: string]: string | undefined;
}
```

```
const object: StringByString = {};

const value = object['nonExistingProp'];
console.log(value); // => undefined
```

3.2 String and number key

Let's say you have a dictionary of number names:

```
interface NumbersNames {
  [key: string]: string
}

const names: NumbersNames = {
  '1': 'one',
  '2': 'two',
  '3': 'three',
  // etc...
};
```

Accessing a value by a string key works as expected:

```
const value1 = names['1']; // OK
```

Would it be an error if you access a value by a number `1`?

```
const value2 = names[1]; // OK
```

Nope, all good!

JavaScript implicitly coerces numbers to strings when used as keys in property accessors (`names[1]` is the same as `names['1']`). TypeScript performs this coercion too.

You can think that `[key: string]` is the same as `[key: string | number]`.

4. Index signature vs Record

TypeScript has a utility type `Record<Keys, Values>` to annotate records, similar to the index signature.

```
const object1: Record<string, string> = { prop: 'Value' }; // OK
const object2: { [key: string]: string } = { prop: 'Value' }; // OK
```

The big question is... when to use a `Record<Keys, Values>` and when an index signature? At first sight, they look quite similar!

As you saw earlier, the index signature accepts only `string`, `number` or `symbol` as key type. If you try to use, for example, a union of string literal types as keys in an index signature, it would be an error:

```
interface Salary {
  // Type error:
  // An index signature parameter type cannot be a literal type or ger
  // Consider using a mapped object type instead.
  [key: 'yearlySalary' | 'yearlyBonus']: number
}
```

This behavior suggests that *the index signature is meant to be generic in regards to keys*.

But you can use a union of string literals to describe the keys in a `Record<Keys, Values>`:

```
type SpecificSalary = Record<'yearlySalary'|'yearlyBonus', number>
type GenericSalary = Record<string, number>

const salary1: SpecificSalary = {
  'yearlySalary': 120_000,
  'yearlyBonus': 10_000
}; // OK
```

If you'd like to limit the keys to a union of specific strings, then `Record<'prop1' | 'prop2' | ... | 'propN', Values>` is the way to go instead of an index signature.

5. Conclusion

An index signature annotiation fits well the case when you don't know the exact structure of the object, but you know the key and value types.

The index signature consists of the index name and its type in square brackets, followed by a colon and the value type: `{ [indexName: Keys]: Values }`. `Keys` can be a `string`, `number`, or `symbol`, while `Values` can be any type.

To limit the key type to a specific union of strings, then using the `Record<Keys, Values>` utilty type is a better idea. The index signature doesn't support unions of string literal types.

Check also my post about record type.

Do you prefer index signatures or `Record<Keys, Values>` utility type?

Like the post? Please share!

Suggest Improvement

About Dmitri Pavlutin

Software developer and sometimes writer. My daily routine consists of (but not limited to) drinking coffee, coding, writing, overcoming boredom 😉. Living in the sunny Barcelona. 🇪🇸

---

19 Comments - *powered by utteranc.es*

loopmode commented on 24 Sept 2021

Thanks for yet another good write-up.
My two cents regarding this one:
Sometimes it is useful to just cast the key, here `name`, as `keyof typeof dictionary`.
This does help with (static) data objects, especially with uniform values, when you don't want to (or can't) define a type/interface beforehand, as it's basically just a lookup.

👍 1

panzerdp commented on 24 Sept 2021 Owner

> Thanks for yet another good write-up.
> My two cents regarding this one:
> Sometimes it is useful to just cast the key, here `name`, as `keyof typeof dictionary`.
> This does help with (static) data objects, especially with uniform values, when you don't want to (or can't) define a type/interface beforehand, as it's basically just a lookup.

Yes, you want to use specific keys, then `keyof` is useful.

masiucd commented on 25 Sept 2021

Good post Dimitri as usual

👍 1

panzerdp commented on 25 Sept 2021 Owner

> Good post Dimitri as usual

Thanks @masiucd.

AshNaz87 commented on 25 Sept 2021

Hey Dmitri, I got this:

```
type Indexed<K extends string, V> = {
    [key in K]: V;
};
```

What would you do differently?

jacobvr commented on 29 Sept 2021

Thank you, learned about `Record<Keys, Type>` !

panzerdp commented on 1 Oct 2021 Owner

> Thank you, learned about `Record<Keys, Type>` !

You're welcome @jacobvr!

panzerdp commented on 2 Oct 2021 Owner

> Hey Dmitri, I got this:
>
> ```
> type Indexed<K extends string, V> = {
>     [key in K]: V;
> };
> ```
>
> What would you do differently?

@AshNaz87 The Indexed must also support `number` and `symbol` as key.

HugaidaS commented on 4 Oct 2021

Thanks for this article!Very informative!

panzerdp commented on 4 Oct 2021 Owner

> Thanks for this article!Very informative!

@Ellinsa Thanks Victoria!

joquijada commented on 9 Nov 2021

Nice post, it cleared up certain things for me.

panzerdp commented on 16 Nov 2021 Owner

> Nice post, it cleared up certain things for me.

Thanks @joquijada

Thanks @joquijada.

Super-Chama commented on 11 Feb 2022

Thanks for a great article, this was helpful! but in my case I wanted Record types to have fixed type for value but be able to flexibly define indexes. This is how i solved it.

```
type Config<T extends string | number | symbol> = Record<T, number>
```

This give me ability to define indexes on fly,

```
type Colors = Config<'Red' | 'Green' | 'Blue'>;

const colors: Colors = {
  Red: 5,
  Green: 5,
  Blue: 5,
};
```

👍 1

riteshwaghela commented on 3 Oct 2022

very Helpful. Thanks for the writeup.

amaben2020 commented on 4 Oct 2022

Great writeup DP, your site is awesome.

panzerdp commented on 4 Oct 2022 Owner

@riteshwaghela @amaben2020 Glad you find it helpful!

gitnupur commented on 4 May 2023

Thanks a ton. Another great article. Very helpful.

panzerdp commented on 4 May 2023 Owner

> Thanks a ton. Another great article. Very helpful.

You're welcome @gitnupur.

klocus commented on 26 Jul 2023

What if we want the key to be able to name only according to strict definitions, but at the same time we do not know which keys we will use? Nothing more simple than that!

```
const object: { [key in MyCustomType]?: number } = {};
```

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub