JUNE 15, 2020 / #TYPESCRIPT

Advanced TypeScript Types Cheat Sheet (with Examples)

Ibrahima Ndaw

TypeScript is a typed language that allows you to specify the type of variables, function parameters, returned values, and object properties.

Here an advanced TypeScript Types cheat sheet with examples.

Let's dive in.

Intersection Types

An intersection type is a way of combining multiple types into one. This means that you can merge a given type A with a type B or more and get back a single type with all properties.

```
type LeftType = {
  id: number
  left: string
}

type RightType = {
  id: number
  right: string
}

type IntersectionType = LeftType & RightType

function showType(args: IntersectionType) {
  console.log(args)
}

showType({ id: 1, left: "test", right: "test" })
// Output: {id: 1, left: "test", right: "test"}
```

As you can see, `IntersectionType` combines two types - `LeftType` and `RightType` and uses the `&` sign to construct the intersection type.

Union Types

Union types allow you to have different types annotation within a given variable.

```
type UnionType = string | number

function showType(arg: UnionType) {
  console.log(arg)
}

showType("test")
// Output: test

showType(7)
// Output: 7
```

Generic Types

A generic type is a way of reusing part of a given type. It helps to capture the type `T` passed in as a parameter.

```
function showType<T>(args: T) {
  console.log(args)
}

showType("test")
// Output: "test"

showType(1)
// Output: 1
```

To construct a generic type, you need to use the brackets and pass `T` as a parameter. Here, I use `T` (the name is up to you) and then, call the function `showType` twice with different type annotations because it's generic - it can be reused.

```
interface GenericType<T> {
  id: number
  name: T
}

function showType(args: GenericType<string>) {
  console.log(args)
}

showType({ id: 1, name: "test" })
// Output: {id: 1, name: "test"}

function showTypeTwo(args: GenericType<number>) {
  console.log(args)
}

showTypeTwo({ id: 1, name: 4 })
// Output: {id: 1, name: 4}
```

Here, we have another example that has an interface `GenericType` which receives a generic type `T`. And since it's reusable, we can call it first with a string and then a number.

```
function showType(args: GenericType<number, string>) {
  console.log(args)
}

showType({ id: 1, name: "test" })
// Output: {id: 1, name: "test"}

function showTypeTwo(args: GenericType<string, string[]>) {
  console.log(args)
}

showTypeTwo({ id: "001", name: ["This", "is", "a", "Test"] })
// Output: {id: "001", name: Array["This", "is", "a", "Test"]}
```

A generic type can receive several arguments. Here, we pass in two parameters: `T` and `U`, and then use them as type annotations for the properties. That said, we can now use the interface and provide different types as arguments.

Utility Types

TypeScript provides handy built-in utilities that help to manipulate types easily. To use them, you need to pass into the `<>` the type you want to transform.

Partial

- `Partial<T>`

Partial allows you to make all properties of the type `T` optional. It will add a `?` mark next to every field.

```
interface PartialType {
  id: number
  firstName: string
  lastName: string
}

function showType(args: Partial<PartialType>) {
  console.log(args)
}

showType({ id: 1 })
// Output: {id: 1}
```

As you can see, we have an interface `PartialType` which is used as type annotation for the parameters received by the function `showType()`. And to make the properties optional, we have to use the `Partial` keyword and pass in the type `PartialType` as an argument. That said, now all fields become optional.

Required

- `Required<T>`

Unlike `Partial`, the `Required` utility makes all properties of the type `T` required.

```
interface RequiredType {
  id: number
  firstName?: string
  lastName?: string
}

function showType(args: Required<RequiredType>) {
  console.log(args)
}

showType({ id: 1, firstName: "John", lastName: "Doe" })
// Output: { id: 1, firstName: "John", lastName: "Doe" }

showType({ id: 1 })
// Error: Type '{ id: number: }' is missing the following properties from type 'Required<Required
```

The `Required` utility will make all properties required even if we make them optional first before using the utility. And if a property is omitted, TypeScript will throw an error.

Readonly

- `Readonly<T>`

This utility type will transform all properties of the type `T` in order to make them not reassignable with a new value.

```
interface ReadonlyType {
  id: number
  name: string
```

```
}

showType({ id: 1, name: "Doe" })
// Error: Cannot assign to 'id' because it is a read-only property.
```

Here, we use the utility `Readonly` to make the properties of `ReadonlyType` not reassignable. That said, if you try to give a new value to one of these fields, an error will be thrown.

Besides that, you can also use the keyword `readonly` in front of a property to make it not reassignable.

```
interface ReadonlyType {
  readonly id: number
  name: string
}
```

Pick

- `Pick<T, K>`

It allows you to create a new type from an existing model `T` by selecting some properties `K` of that type.

```
interface PickType {
  id: number
  firstName: string
  lastName: string
}

function showType(args: Pick<PickType, "firstName" | "lastName">) {
  console.log(args)
}

showType({ firstName: "John", lastName: "Doe" })
// Output: {firstName: "John"}

showType({ id: 3 })
// Error: Object literal may only specify known properties, and 'id' does not exist in type 'Pick
```

Omit

- `Omit<T, K>`

The `Omit` utility is the opposite of the `Pick` type. And instead of selecting elements, it will remove `K` properties from the type `T`.

```
interface PickType {
  id: number
  firstName: string
  lastName: string
}

function showType(args: Omit<PickType, "firstName" | "lastName">) {
  console.log(args)
}

showType({ id: 7 })
// Output: {id: 7}

showType({ firstName: "John" })
// Error: Object literal may only specify known properties, and 'firstName' does not exist in typ
```

This utility is similar to the way `Pick` works. It expects the type and the properties to omit from that type.

Extract

- `Extract<T, U>`

`Extract` allows you to construct a type by picking properties that are present in two different types. The utility will extract from `T` all properties that are assignable to `U`.

```
interface FirstType {
  id: number
  firstName: string
  lastName: string
}

interface SecondType {
  id: number
```

```
// Output: "id"
```

Here, we have two types that have in common the property `id`. And hence by using the `Extract` keyword, we get back the field `id` since it's present in both interfaces. And if you have more than one shared field, the utility will extract all similar properties.

Exclude

Unlike `Extract`, the `Exclude` utility will construct a type by excluding properties that are already present in two different types. It excludes from `T` all fields that are assignable to `U`.

```
interface FirstType {
  id: number
  firstName: string
  lastName: string
}

interface SecondType {
  id: number
  address: string
  city: string
}

type ExcludeType = Exclude<keyof FirstType, keyof SecondType>

// Output; "firstName" | "lastName"
```

As you can see here, the properties `firstName` and `lastName` are assignable to the `SecondType` type since they are not present there. And by using the `Extract` keyword, we get back these fields as expected.

Record

- `Record<K,T>`

This utility helps you to construct a type with a set of properties `K` of a given type `T`. `Record` is really handy when it comes to mapping the properties of a type to another one.

```
interface EmployeeType {
  id: number
```

```
  0: { id: 1, fullname: "John Doe", role: "Designer" },
  1: { id: 2, fullname: "Ibrahima Fall", role: "Developer" },
  2: { id: 3, fullname: "Sara Duckson", role: "Developer" },
}

// 0: { id: 1, fullname: "John Doe", role: "Designer" },
// 1: { id: 2, fullname: "Ibrahima Fall", role: "Developer" },
// 2: { id: 3, fullname: "Sara Duckson", role: "Developer" }
```

The way `Record` works is relatively simple. Here, it expects a `number` as a type which is why we have 0, 1, and 2 as keys for the `employees` variable. And if you try to use a string as a property, an error will be thrown. Next, the set of properties is given by `EmployeeType` hence the object with the fields id, fullName, and role.

NonNullable

- `NonNullable<T>`

It allows you to remove `null` and `undefined` from the type `T`.

```
type NonNullableType = string | number | null | undefined

function showType(args: NonNullable<NonNullableType>) {
  console.log(args)
}

showType("test")
// Output: "test"

showType(1)
// Output: 1

showType(null)
// Error: Argument of type 'null' is not assignable to parameter of type 'string | number'.

showType(undefined)
// Error: Argument of type 'undefined' is not assignable to parameter of type 'string | number'.
```

Here, we pass the type `NonNullableType` as an argument to the `NonNullable` utility which constructs a new type by excluding `null` and `undefined` from that type. That said, if you pass a nullable value, TypeScript will throw an error.

Mapped types

Mapped types allow you to take an existing model and transform each of its properties into a new type. Note that some utility types covered earlier are also mapped types.

```
type StringMap<T> = {
  [P in keyof T]: string
}

function showType(arg: StringMap<{ id: number; name: string }>) {
  console.log(arg)
}

showType({ id: 1, name: "Test" })
// Error: Type 'number' is not assignable to type 'string'.

showType({ id: "testId", name: "This is a Test" })
// Output: {id: "testId", name: "This is a Test"}
```

`StringMap<>` will transform whatever types that passed in into a string. That said, if we use it in the function `showType()`, the parameters received must be a string - otherwise, an error will be thrown by TypeScript.

Type Guards

Type Guards allow you to check the type of a variable or an object with an operator. It's a conditional block that returns a type using `typeof`, `instanceof`, or `in`.

- `typeof`

```
function showType(x: number | string) {
  if (typeof x === "number") {
    return `The result is ${x + x}`
  }
  throw new Error(`This operation can't be done on a ${typeof x}`)
}

showType("I'm not a number")
// Error: This operation can't be done on a string

showType(7)
// Output: The result is 14
```

condition.

- `instanceof`

```
class Foo {
  bar() {
    return "Hello World"
  }
}

class Bar {
  baz = "123"
}

function showType(arg: Foo | Bar) {
  if (arg instanceof Foo) {
    console.log(arg.bar())
    return arg.bar()
  }

  throw new Error("The type is not supported")
}

showType(new Foo())
// Output: Hello World

showType(new Bar())
// Error: The type is not supported
```

Like the previous example, this one is also a type guard that checks if the parameter received is part of the `Foo` class or not and handles it consequently.

- `in`

```
interface FirstType {
  x: number
}
interface SecondType {
  y: string
}

function showType(arg: FirstType | SecondType) {
  if ("x" in arg) {
    console.log(`The property ${arg.x} exists`)
    return `The property ${arg.x} exists`
  }
```

```
showType({ y: "ccc" })
// Error: This type is not expected
```

The `in` operator allows you to check whether a property `x` exists or not on the object received as a parameter.

Conditional Types

Conditional types test two types and select one of them depending on the outcome of that test.

```
type NonNullable<T> = T extends null | undefined ? never : T
```

This example of the `NonNullable` utility type checks if the type is null or not and handle it depending on that. And as you can note, it uses the JavaScript ternary operator.

Thanks for reading.

You can find other great content like this on my blog or follow me on Twitter to get notified.

Ibrahima Ndaw

JavaScript enthusiast, Full-stack developer & blogger
