Aliases with Type

In this lesson, we will see how to create aliases for types.

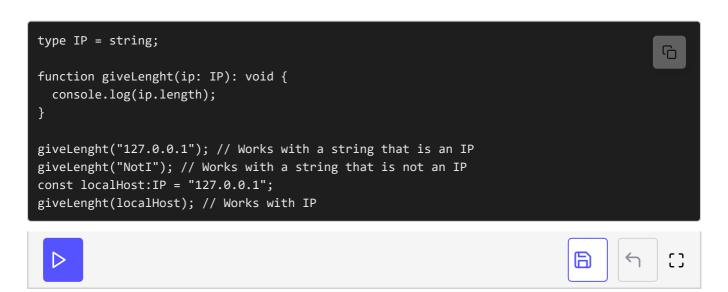
WE'LL COVER THE FOLLOWING ^

- Type aliases for primitives
- Type aliases for unions
- Alias with a function type

Type aliases are the notion shortcut names for existing types, or many types under a single name. The goal of creating an alias is to increase the readability of the code as well as to have a reusable and common way to identify a type.

Type aliases for primitives

At its simplest, an alias can represent a single primitive. The alias will have the same properties as the underlying primitive type. The following example shows that the function takes an IP as the type, which is a string at line 3. The difference is that the readability of the code is better by self-documenting what we expect of that string to be – an IP address.



well as a string that is an invalid IP at **line 8** and also works (as expected) with the type IP.

Type aliases for unions

An everyday use case is to gather many primitive types under a type alias. This avoids repeating the union over and over.

For example, instead of writing:

```
function setId(id: number | string | null): void { }
function getId(): number | string | null{ return null; }
function validId(id: number | string | null): void { }
```

You could write:

```
type ID = number | string | null;
function setId(id: ID): void { }
function getId(): ID { return null; }
function validId(id: ID): void { }
```

Not only does this have less repetition, it is also clearer about the ID and is easier to modify the type later. For example, removing null at line 1 would be a single line change instead of many in the following code:

```
type ID = number | string | null;
function setId(id: ID): void { }
function getId(): ID { return null; }
function validId(id: ID): void { }
```

Alias with a function type

A type alias can be used for an array, other types, and even functions. If you are using a function in a parameter and it needs a specific arrangement of parameters, it is often clearer to use an alias.

Instead of:

```
function execute(code: (id: number, name: string) => boolean, error: (message: string) => voi
    if (!code(1, "Name1")) {
        error("Does not work");
    }
}

const myAlgorithm: (id: number, name: string) => boolean = (id: number, name: string): boolear
    return false;
}

const errorHandling: (message: string) => void = (message: string): void => {
        console.log(message);
}

execute(myAlgorithm, (errorHandling));
```

You could write with a function type alias that are defined at **line 1-2** in the following example. Compare the previous example **line 1** with the next **line 3**. The next example should be easier to read comparing to the previous example.

```
type Code = (id: number, name: string) => boolean;
type ErrorCallback = (message: string) => void;
function execute(code: Code, error: ErrorCallback): void {
   if (!code(1, "Name1")) {
      error("Does not work");
   }
}
const myAlgorithm: Code = (id: number, name: string): boolean => {
   return false;
}
const errorHandling: ErrorCallback = (message: string): void => {
      console.log(message);
}
execute(myAlgorithm, (errorHandling));
```

It is also possible to mix primitives and functions. Take a look at **line 3**. A use case that we see often with API is when a name can be provided or a dynamic way to find the name can be used.

```
type DynamicName = () => string;
type NameFinder = Name | DynamicName; // Primitive and a function
function getName(name: NameFinder): Name {
   if (typeof name === "string") {
      return name;
   }
   else {
      return name();
   }
}
console.log(getName("Name1"));
console.log(getName(() => { return Math.random() > 0.5 ? "Random1" : "Random2" }));
```

