Overload Functions to Enrich your Definition

In this lesson we discuss about enriching functions with several signatures of overload.

WE'LL COVER THE FOLLOWING

The reason to use overloaded functions

TypeScript allows having an overload of a function. Often, the term overload is linked to an object-oriented concept which you will see later. Any function, even if not in a class, can have an overload. The reason to use overload functions is that you may have a function that takes some required parameters only in a specific scenario. Instead of using the overloaded functions, it's possible to use optional parameters and default parameters.

The advantage of overloaded functions is clarity for the consumer. A consumer perspective will see several functions with different signatures (parameters and return types). However, it's still a single function.

Overloading return types

Overloading affects not just parameters but also the return type. Overloading is quite useful since without it you wouldn't know which group of optional parameters belong together and which of them match which return type. Using overloading requires having several functions with the same name defined next to each other with a body. That's right, a signature from the keyword function to the return type that ends with a semi-colon. Only the last defined function requires being more conventional with a full body. This last one must contain the aggregation of all the overloads.

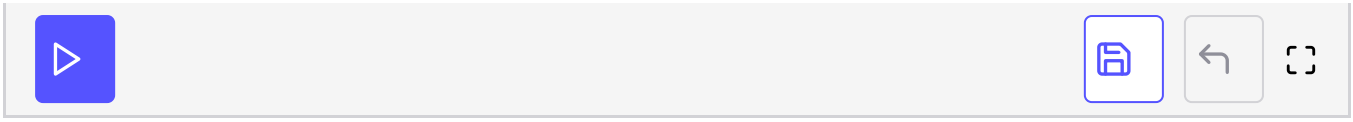
Example

For example, if you have an overload that takes a number and returns a number, and the second one that takes a string and returns a string, the last definition will contain a signature with a single parameter that unites a number and a string with a return type of number union string.

```
function f1(p1:number):number;
function f1(p1:string):string;
function f1(p1:number|string):number|string;
```

The last line handles all function overloads which means that you will see at the beginning of the function code that it will assess if specific parameters are defined and act accordingly.

```
function functionWithOverload(param1: number): boolean;
function functionWithOverload(param1: number, param2: string): string;
function functionWithOverload(param1: number, param2: string = "default", param3?: string): b
    if (param3 === undefined) {
        return "string";
    }
    return true;
}
console.log(functionWithOverload(1));
```



Other than providing a clear idea of what parameters are optional and with what return type they are associated, overload syntax makes the Intellisense richer with an IDE that can separate the scenarios with proper documentation and the possible combinations of syntax.

The example above prints “string” because the overload is the first one meaning that `param3`, which is optional, is `undefined` at **line 3**. The `undefined` value leads execution into the condition that returns a `string` instead of a `boolean` value.