

Typescript essentials. Functions

In this lesson, we will review working with functions in Typescript. To define function in Typescript we use the keyword “function”. As an example, let us define function that returns a sum of two input parameters (fig. 1). However, now it can accept not only numbers, but also other data types. Depending on this, function results also can be different.

```
1  function add(a, b) {  
2    |    return a + b;  
3  }  
4  
5  let addRes = add(1, 2);  
6  console.log(addRes);  
7  
8  console.log(add("1", "2"));|
```

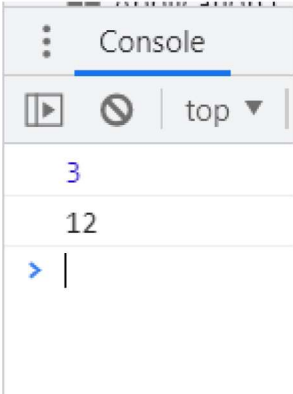


Figure 1 – Function with no data types for parameters and output

If we want our function to have an information about how to use it in a right way, we can give it's parameters and also it's result some types. Previous example can be changed to operate only with numbers (fig. 2).

```
10 function addNumbers(a:number, b: number): number {  
11   |    return a + b;  
12 }  
13  
14 const a = 5, b = 3;  
15 console.log(addNumbers(a, b));
```

Figure 2 – Function for adding numbers

Now function cannot be used in wrong way and we cannot pass to it not numeric parameters or assign it's output with nonnumeric variable (fig. 3).

```

16 console.log(addNumbers("3", "2"));
    Argument of type 'string' is not assignable to parameter of type
    'number'. ts(2345)
    View Problem (Alt+F8) No quick fixes available

18 let addOutput: string;
19 addOutput = addNumbers(a, b)
    let addOutput: string

```

Figure 3 – Tries to operate with typed function in a wrong way

Function in Typescript can be also defined as a variable. It will be useful little bit later when we start to work with classes and interfaces, but now let's simply look how it works. First, we need to define a variable with function data type. On this step, we describe how many parameters function will have and what will be their data types. Other words, this is a pattern that define function type, or function signature. In the next example, it is defined two variable, which will be functions. First function variable “addNumbersVariable” has two number parameters and return number data type. Second function variable “showStringVariable” has one string parameter and returns nothing.

At the next step, we create “addNumbersFunction” function that has defined according to “addNumbersVariable” signature. According to this fact, next we can assign “addNumbersFunction” to variable “addNumbersVariable”. We can also do it without separate named function, by creating a function directly into goal variable as it was done with “showStringVariable” variable. This approach is called anonymous function (fig. 4).

```

1 let addNumbersVariable: (x: number, y: number) => number;
2 let showStringVariable: (s: string) => void;
3
4 function addNumbersFunction (x: number, y: number): number {
5     return x + y;
6 }
7
8 addNumbersVariable = addNumbersFunction;
9 console.log(addNumbersVariable(5, 10));

```

```

10
11 showStringVariable = function(message: string): void {
12     console.log(message);
13 }
14 showStringVariable('Hello from showStringVariable');|

```

Figure 4 – Working with variable of function datatype

While working with function parameters there are two additional possibilities: default values and optional parameters. When we use one of those, it is not needed to obligatory pass such a parameter to function. Let's look to an example of the function with two string parameters that are optional (fig. 5). As we can see, such a function can be called with two parameters, with one parameter and even with no parameters ether.

To define parameter as optional we need to add question symbol “?” after variable name. And if we use optional parameters they must follow after obligatory parameters.

```

1  function funWithOptPar(name?: string, surname?: string) {
2      let person = "";
3      if (name) {
4          person += name;
5      }
6      if (surname) {
7          person += ' ' + surname;
8      }
9      if (person == '') {
10         person = 'guest';
11     }
12     return person;
13 }
14
15 console.log(funWithOptPar('Mikhaylo', 'Dvoretskyi'));
16 console.log(funWithOptPar('Mikhaylo'));
17 console.log(funWithOptPar());|

```

Figure 5 – Function with optional parameters

Beside optional parameters, default parameter values can be used. For instance, previous example can use default values to achieve the same result (fig. 6). To define such a parameter we need to put a value in it after its declaration.


```

19 function funWithDefPar(name: string = '', surname: string = '') {
20     let person = `${name} ${surname}`;
21
22     if (person.trim() == '') {
23         person = 'guest';
24     }
25     return person;
26 }
27
28 console.log(funWithDefPar('Mikhaylo', 'Dvoretskyi'));
29 console.log(funWithDefPar('Mikhaylo'));
30 console.log(funWithDefPar());

```

Figure 6 – Function with default parameter value

Typescript maintain functions with undefined quantity of parameters. To create such a function we need to use rest parameter. It is a parameter, that in function signature is an array, and parameter name starts with ellipsis “...”. This parameters must follow at the end of parameters list. For instance, let’s define function that will calculate sum of numbers, which passed as function parameters (fig. 7).

```

1  function sumFun(... params: number[]): number {
2      let sumVar = 0;
3      params.forEach(
4          function(elem) {
5              sumVar += elem;
6          }
7      )
8      return sumVar;
9  }
10
11 console.log(sumFun(1,2,3,4));

```

Figure 7 – Sum function with rest parameter

Now let’s take a look at function context concept. Function context is an environment where the function was called. And it is located in «this» link. To figure it out more smoothly we define function “contextReporter” that will output “this” link to the console. If we call it in global context we receive global object “window” (fig. 8).

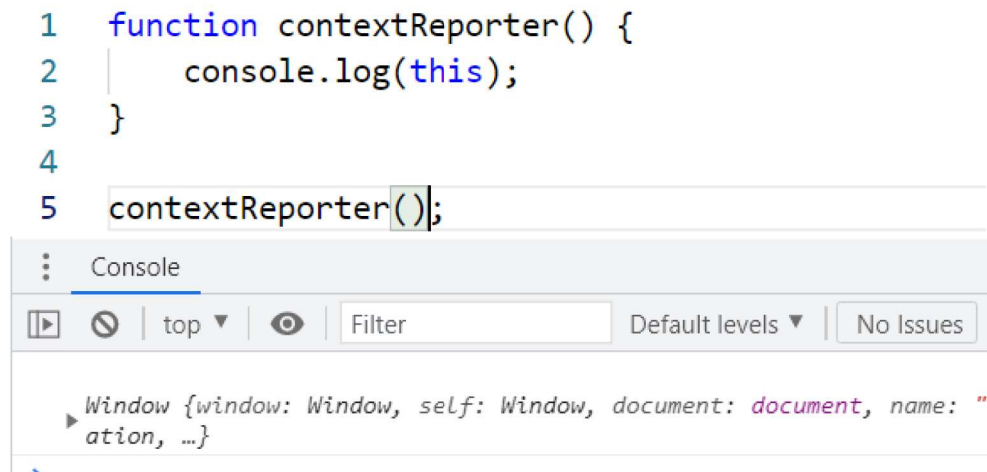


Figure 8 – Function “myContextReporter”

Then let’s define an object with method that is a link for “contextReporter” function. If we call it, as an object method, the function context will be different from the previous example (fig. 9).

```
7 let contextObj = {  
8   objField: 'Field 1',  
9   objMethod: contextReporter  
10 };  
11  
12 contextObj.objMethod();
```

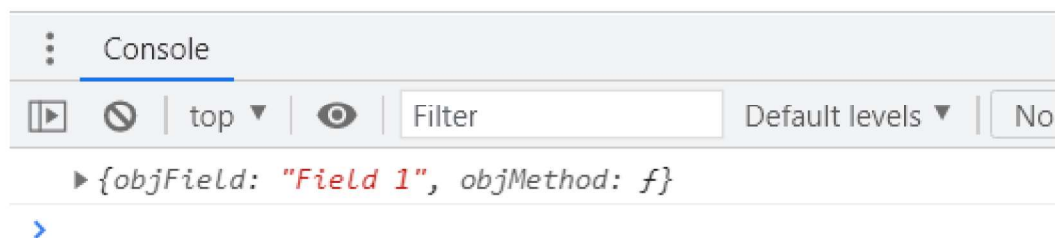


Figure 9 – Call function in object context

In JavaScript we also can manipulate with context by several different approaches. These are “call”, “apply” and “bind” object methods (fig. 10).

```
14 contextReporter.call(contextObj);  
15 contextReporter.apply(contextObj);  
16  
17 let contextNewFun = contextReporter.bind(contextObj);  
18 contextNewFun();
```

Figure 10 – Setting function context

In typescript arrow function also can be used. In the next example, we declare several variables that are initialized as function but in different ways. First one is initialized as common function. We can simplify this syntax by using an arrow function approach that has been shown in second example. We don't write "function" keyword and use an arrow "=>" to separate signature from function body. Next, we simplify it even more by excluding "return" keyword. It can be done only if function contain only one row. And, finally we can maximally simplify it by excluding also brackets around parameters and its data types (fig. 11).

```
1  let myFunVariable1 = function(x: number) {  
2    |   return x * x;  
3  }  
4  
5  let myFunVariable2 = (x: number) => {  
6    |   return x * x;  
7  }  
8  
9  let myFunVariable3 = (x: number) => x * x;  
10  
11 let myFunVariable4 = x => x * x;
```

Figure 11 – Arrow function examples

One of the difference of last arrow function example is that it's parameter has lost data type and become "any" data type. To avoid it we can define function signature before initialize it (fig. 12).

```
13 | let myFunVariable5: (number) => number = x => x * x;
```

Figure 12 – Defining function signature before initialize it with an arrow function

So, as we can see, in case of using anonymous functions, arrow functions allows to reduce code amount and makes it more understandable. Another arrow function advantage is working with context it little bit different way. Let's illustrate it on example. Firstly, we define a function "repeatFunction" that can run some other callback function several times. This function has two parameters – number of

execution times, and a callback function that will be executed (fig. 13). Then we define an object with two methods that will call previously defined function “repeatFunction”. One of the methods will pass common function as a callback, and another will pass arrow function. And both will output “this” context on the console.

```
1  function repeatFunction(howManyTimes: number, callBack: () => void) {
2      for(let i = 0; i < howManyTimes; i++) {
3          callBack();
4      }
5  }
6
7  let exampleObj = {
8      objName: 'example object',
9      commonFunctionMethod: function() {
10         repeatFunction(2, function() {
11             console.log(this);
12         })
13     },
14     arrowFunctionMethod: function() {
15         repeatFunction(2, () => console.log(this))
16     }
17 }
18
19 exampleObj.commonFunctionMethod();
20 exampleObj.arrowFunctionMethod();
```

Figure 13 – Function that will illustrate difference in context while using arrow function

In case of common function context will be lost and will be shown as global context “window”. Arrow function will save “exampleObj” context (fig. 14).

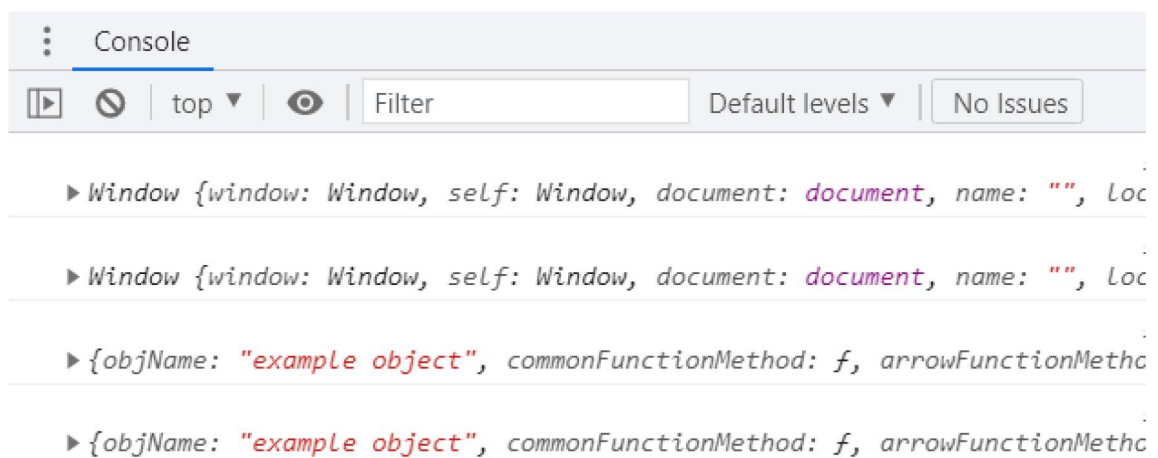


Figure 14 – Context output result