Typescript essentials. Variables

There are two ways to define variable type – explicitly and implicitly. For instance, variables “name” and “surname” with type “string” can be declared, as it is given in fig. 1.

```
1   let firstName: string;
2   firstName = 'John'
3   let surName: string = 'Smith'
```

Figure 1 – Variables “name” and “surname” declaration

Variable can also get it’s type implicitly by giving it value (fig. 2).

```
5   let age = 35
```

Figure 2 – Giving variable type through it’s value

Once variable was given a type, it cannot be changed. For example, in JavaScript you can set value of another type to the same variable (fig. 3) but in Typescript it will be an error (fig. 4).

```
5   let age = 35;
6   age = 'thirty five'
```

Figure 3 – Change variable type in JavaScript

```
5   let age = 35
6   age = 'thirty five'
        let age: number
        Type 'string' is not assignable to type 'number'
```

Figure 4 – Error in Typescript while changing variable type

Variable can be defined with keyword “var” or “let”. The main difference is variable scope. For example, let’s make a loop with index variable in two different ways – with “var” and “let” variable. As we can see (fig. 5), “var” variable is visible after loop ends, so it is function scope. If we try to access “let” variable from outside the loop, we have an error.

```
1  for(var i = 0; i < 10; i ++) {
2      var test = i;
3  }
4
5  console.log(i);
6  console.log(test);
7
8  for(let i2 = 0; i2 < 10; i2 ++) {
9      let test2 = i2;
10 }
11
12 console.log(i2);
13 console.log(test2);
```

```
any
Cannot find name 'i2'.
```

Figure 5 – "Var" and "let" keywords for variable declaration

Also with keyword "var" we can define variable more than one time. But using "let" keyword, it leads to an error (fig. 6).

```
1  var strExample = ''
2  var strExample = 'my test string'
3
4  let strExample2 = ''
5  let strExample2 = 'my test string'
```

```
let strExample2: string
Cannot redeclare block-scoped variable 'strExample2'
```

Figure 6 – Defining variable more than one time

There is also one more example of "var" in classic JavaScript while working with functions. If we initialize valiable, then define a function where we read this variable, and then initialize it again and then read it again, first time we receive "undefined". That happens because in JS variable declaration in the scope comes first, and only then other part of code is compiled (fig. 7).

```
1  var outer = 10
2
3  function myFun1() {
4      console.log(outer)
5      var outer = 5
6      console.log(outer)
7  }
8
9  myFun1()
```

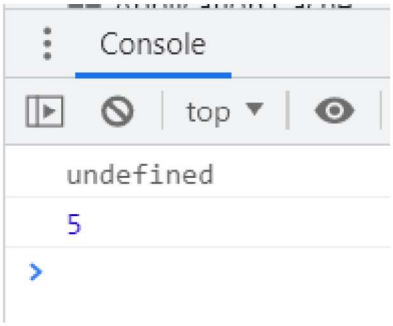

Figure 7 – Unexpected variable value

Besides, the example with setTimeout can illustrate another problem connected with "var". With code in fig. 8 we expected to receive "0,1,2,3,4". But instead we got "5" five times. That is because "i" variable is one for all five loop iterations and it's value is "5" when first setTimeout function is run. If we change "var" to "let" the problem is gone (fig. 9).

```
11  for(var i=0; i<5; i++) {
12      setTimeout(function() {
13          console.log(i), 100
14      })
15  }
```

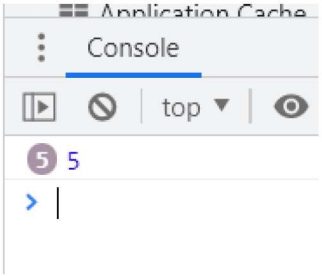

Figure 8 – "Var" with setTimeout

```
11  for(let i=0; i<5; i++) {
12      setTimeout(function() {
13          console.log(i), 100
14      })
15  }
```

Figure 9 – "Let" with setTimeout

It is also a constant can be defined in Typescript. There was no such a thing in pure JS and it allows to define values that cannot be changed next (fig. 10).

```
11  const myConstant: string = "Test constant"
12  myConstant = 'Try to change constant value'
        Cannot assign to 'myConstant' because it is a constant
```

Figure 10 – Define a constant

Let's have a look on different variable types that Typescript provides for variable declaration. If variable is boolean, it can be true, false or undefined (fig. 11).

```
1  let boolExample1: boolean = true
2  let boolExample2: boolean = 1
       let boolExample2: boolean
       Type 'number' is not assignable to type 'boolean'
3  let boolExample3: boolean
4
5  console.log(boolExample3)
```

Figure 11 – Boolean type

Number type can be initialized in various number systems. Among them are binary, octal, decimal and hexadecimal ones. Examples of assigning value "10" to variables using these number systems, are given in fig. 12.

```
7   //  Number
8   let a1_decimal: number = 10;
9   let a2_hex: number = 0x000a
10  let a3_binary: number = 0b1010
11  let a4_octal: number = 0o12;
12
13  console.log(a1_decimal);
14  console.log(a2_hex);
15  console.log(a3_binary);
16  console.log(a4_octal);
```

Figure 12 – Assigning values to number type variables

While assigning value to string, variable concatenation and template strings can be used. Here both strings and numbers can take part (fig. 13). To inject variable value into template we use "$" symbol and then in curly brackets follows variable name.

```
18  // String
19  let firstName1: string = "Ivan";
20  let age1: number = 25;
21  let messageConcat: string = "Hello, my name is " + firstName1 + " I'm " +
    age1 + " years old." // concatenation
22  let messageTemplate: string = `Hello, my name is ${firstName1} I'm ${age1}
    years old.` // template string
23  console.log(messageConcat)
24  console.log(messageTemplate)
```

Figure 13 – Using concatenation and template string

To declare array of variables we need simply to add square brackets after data type []. In fig. 14 is given an example where arrays of number and string data types are assigned. Another syntax of array declaration is usage of generic of array type. We cannot add another data type to such an array and, as we can see, array can be initialized at once after its declaration, as a variable with common data type.

```
1   let months: string[]  = ["January", "February", "March", "April", "May",
    "June", "July", "August", "September", "October", "November", "December"];
2   for(let i = 0; i < months.length; i++) {
3       console.log(months[i]);
4   }
5
6   let numbers: number[];
7   numbers = [1, 2, 3];
8   for(let i = 0; i < numbers.length; i++) {
9       console.log(numbers[i]);
10  }
11
12  let values: Array<number> = [-1, -2, -3]; // generic of Array type
13  for(let i = 0; i < values.length; i++) {
14      console.log(values[i]);
15  }
```

Figure 14 – Array data type

As we can see, arrays are pretty the same to other languages such as Java or C#.

Another data type which is little bit similar to array is tuple. It is an ordered set with fixed length. We can say that it is an array with different data type but also with fixed length. For example, let's create tuple with one number and two strings in it (fig. 15). Also an array of tuples can be defined.

```
let tuple1: [string, string, number]
    = ['John', 'Smith', 35];
console.log(
    `Namae ${tuple1[0]}, surname ${tuple1[1]}, age ${tuple1[2]}`
    )

let students: [string, string][] = []
students.push(['Ann', '208'])
students.push(['Piter', '309'])

for(let i = 0; i < students.length; i++) {
    console.log(`Group ${students[i][0]} name ${students[i][1]}`)
}
```

Figure 15 – Usage of tuples

Tuple can be useful when we need to store in one object several values with different data types. Or, for example, when we need to return several values from a function.

To describe some set of values enumeration, enum type can be used. It is a type that allows to collect a set of numeric values and provide to them some meaningful names. For instance, we can use “enum” to describe colors of traffic light. Also the function which allows or denies to go is defined (fig. 16). With the keyword “enum” we say that new data type is defined. And then, with a curly brackets we define a set of values that will be included in this enumeration.

```
enum traficLightColor {
    Red, Yellow, Green
}

function allowMovement(color: traficLightColor): boolean {
    if (color == traficLightColor.Green) {
        return true;
    }
    return false;
}

console.log(allowMovement(traficLightColor.Red))
console.log(allowMovement(traficLightColor.Green))
```

Figure 16 – Example of “enum” usage

By default compiler gives a numeric value to each element of new enumeration. We can see it if we put the mouse arrow over one of the “enum” elements. Also we can put “enum” element to the console (fig. 17).

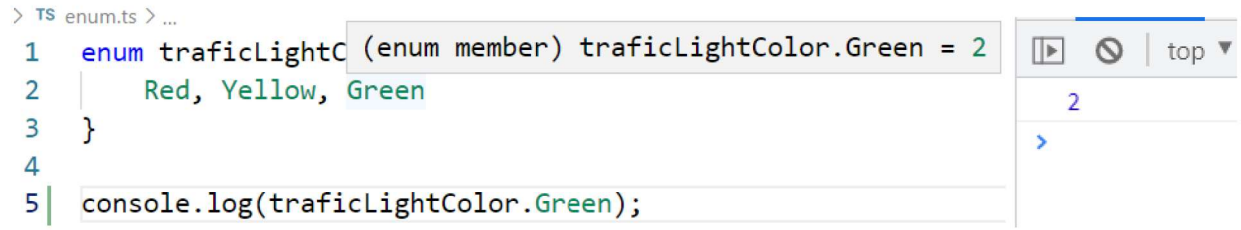

Figure 17 – Number value of enumeration elements

To declare variable, which type is unknown while the application is being developed, the “any” data type can be used. In other words variable with “any” data type in Typescript equivalent to variable declaration it JavaScript. It can be handy if data comes from user or for example from http request. It will not be any errors when we put different data types in such a variable. However, it is always some danger to call a method that is not available for current value (fig. 18). And compiler won’t help us while we are writing our code (fig. 19).

```
1  let anyValue: any = "Any type"; // string
2  anyValue = false; // boolean
3  anyValue = 100; // number
4  console.log(anyValue); // 100
5
6  anyValue.toFixed(); // ok
7  anyValue = "test string value";
8  anyValue.toFixed(); //error
```

```
100                                                    any-type.js:4
▶ Uncaught TypeError: someValue.toFixed is not a function   any-type.js:7
    at any-type.js:7
>
```

Figure 18 – Any data type

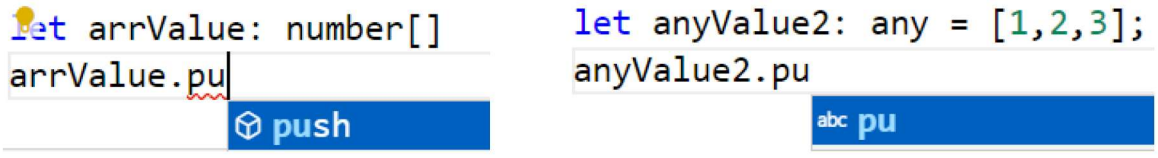

Figure 19 – No hints with “any” data type

There is also a void data type, which allows to define, that function does not return any value. It also can be used to define a variable data type but such a variable can only get “undefined” or “null” values (fig. 20).

```
1  function exampleFunctionOnReturn() : void {
2      alert("hello from function with no return");
3  }
4
5  let voidVariable: void = undefined;
```

Figure 20 – Void data type

Of course, in some cases we can assert variable to certain data type. With type assertion we can tell compiler, which data type we are working with. We can use generics and angle brackets for this purpose. On the other hand, it can be done using “as” keyword (fig. 21). This approach is useful while we are sure that for example one of data attributes, which came from http response, has certain data type or method. Or, we use some JS library that does not provide any types for its output values.

```
1  let someAnyData: any = "Any data";
2  let number1: number = (<string>someAnyData).length;
3  let number2: number = (someAnyData as string).length;
```

Figure 21 – Data type assertion

However, of course we need to remember, that if “someAnyData” variable value won’t be a string data type, we will have an error, or at least unexpected result, while executing this code (fig. 22).

```
1  let someAnyData: any = 12; //"Any data";
2  let number1: number = (<string>someAnyData).length;
3  let number2: number = (someAnyData as string).length;
4  console.log(number1);
```

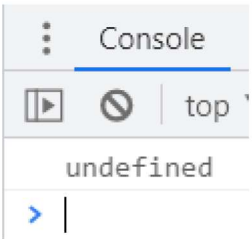

Figure 22 – Data type assertion errors