✓ 100 XP

Create functions in TypeScript

5 minutes

In JavaScript, function definitions don't specify data types for parameters, perform type checking on the passed arguments, or check the number of arguments received. Therefore, you must add the logic for checking these parameters to your functions.

TypeScript simplifies the development of functions and makes them easier to troubleshoot by enabling you to type parameters and return values. TypeScript also adds new options for parameters. For example, while all parameters are optional in JavaScript functions, you can choose to make parameters required or optional in TypeScript.

Adding types to functions helps prevent you from passing values that you shouldn't pass to your functions. Typed functions are especially important when you're working with larger code bases or functions developed by others. While adding types is a simple difference, it offers the benefit of type checking the values that you pass to the function and what is returned. Instead of having to add all the logic to the function to verify the correct value type has been passed to it and the return value is correct, TypeScript helps ensure the correct value types as you develop your code. In addition, when creating the function logic you'll have full autocomplete support, as your editor will know the datatype of the parameters - something JavaScript is typically unable to detect. Autocomplete is especially helpful when you're using functions developed by others because TypeScript clarifies the required input and output types.

As in JavaScript, you can define functions in TypeScript several different ways. Let's look at how these functions differ with the addition of types in TypeScript.

Named functions

A named function is a function declaration written with the `function` keyword and provided with a distinct name within the current scope. Named function declarations are loaded into the execution context before any code runs. This process is known as hoisting, and it means that you can use the function before you declare it.

The syntax for declaring a named function in TypeScript is the same as defining one in JavaScript. The only difference with TypeScript is that you can provide a type annotation for the function's parameters and return value.

This function accepts two parameters of type `number` and returns a `number`.

TypeScript

```typescript
function addNumbers (x: number, y: number): number {
   return x + y;
}
addNumbers(1, 2);
```

Anonymous functions

A **function expression** (or **anonymous function**) is a function that isn't pre-loaded into the execution context, and only runs when the code encounters it. Function expressions are created at runtime and must be declared before they can be called. They aren't hoisted, unlike named function declarations that are hoisted as soon as program execution begins and can be called before their declaration.

Function expressions represent values so they're usually assigned to a variable or passed to other functions, and can be anonymous, meaning the function has no name.

This example assigns a `function` expression to the variable `addNumbers`. Notice that function appears in place of the function name, making the function anonymous. You can now use this variable to call the function.

TypeScript

```typescript
let addNumbers = function (x: number, y: number): number {
   return x + y;
}
addNumbers(1, 2);
```

The following example shows what the named function `sum` looks like when written as an anonymous function. Notice that the name `sum` has been replaced with the keyword `function` and the function has been implemented as an expression in a variable declaration.

TypeScript

```typescript
let sum = function (input: number[]): number {
    let total: number =  0;
    for(let i = 0; i < input.length; i++) {
        if(isNaN(input[i])) {
            continue;
        }
        total += Number(input[i]);
    }
    return total;
}

console.log(sum([1, 2, 3]));
```

As before, you'll get type checking and Intellisense when you use anonymous functions. You'll also notice in this example that the variable `sum` isn't typed, but TypeScript is able to determine its type through something called "contextual typing", a form of type inference. Contextual typing can reduce the amount of effort required to keep your program typed.

Arrow functions

Arrow functions (also called Lambda or fat arrow functions because of the `=>` operator used to define them) provide shorthand syntax for defining an anonymous function. Due to their concise nature, arrow functions are often used with simple functions and in some event handling scenarios.

This example compares the syntax of an anonymous `function` to a single line arrow function. The arrow function abbreviates the syntax by omitting the function keyword and adding the `=>` operator between the parameters and the function body.

TypeScript

```
// Anonymous function
let addNumbers1 = function (x: number, y: number): number {
   return x + y;
}

// Arrow function
let addNumbers2 = (x: number, y: number): number => x + y;
```

In this example, notice also that the curly braces have been removed and there's no `return` statement. Single line arrow functions can use **concise body syntax**, or **implicit return**, which allows the omission of the curly brackets and the `return` keyword.

If the function body has more than a single line, enclose it in curly braces and include the `return` statement (if appropriate.) This example shows what the anonymous function in the earlier example looks like when written as an arrow function.

TypeScript

```
let total2 = (input: number[]): number => {
    let total: number =  0;
    for(let i = 0; i < input.length; i++) {
        if(isNaN(input[i])) {
            continue;
        }
        total += Number(input[i]);
    }
    return total;
}
```

Tip

Arrow functions were introduced in ES2015 so not all browsers support them. By using TypeScript, you can take advantage of these function types and then transpile down to earlier JavaScript versions, if necessary, so your code will work with older browsers.

Next unit: Exercise - Create functions

Continue ›

How are we doing? ☆ ☆ ☆ ☆ ☆