100 XP

Fun with parameters

5 minutes

The TypeScript compiler assumes, by default, that all parameters defined in a function are required. When a function is called, the TypeScript compiler verifies:

- A value has been provided for each parameter.
- Only parameters that the function requires are passed to it.
- The parameters are passed in the order in which they're defined in the function.

These requirements are different from JavaScript. JavaScript assumes that all parameters are optional and allows you to pass more (or fewer) arguments to the function than are defined by it.

In addition to required parameters, you can define functions with optional, default, and rest parameters, as well as deconstructed object parameters.

Required parameters

All function parameters are required, unless otherwise specified, and the number of arguments passed to a function must match the number of required parameters the function expects.

In this example, all parameters are required.

TypeScript

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

addNumbers(1, 2); // Returns 3
addNumbers(1);    // Returns an error
```

Optional parameters

You can also define optional parameters by appending a question mark (?) to the end of the

parameter name.

In this example, `x` is required and `y` is optional. The optional parameter must come after any required parameters in the parameter list. Also, for this function to return the correct value, you must address the possibility that `y` may be passed in as undefined.

TypeScript

```
function addNumbers (x: number, y?: number): number {
    if (y === undefined) {
        return x;
    } else {
        return x + y;
    }
}

addNumbers(1, 2); // Returns 3
addNumbers(1);    // Returns 1
```

Default parameters

You can also assign default values to optional parameters. If a value is passed as an argument to the optional parameter, that value will be assigned to it. Otherwise, the default value will be assigned to it. As with optional parameters, default parameters must come after required parameters in the parameter list.

In this example, `x` is required and `y` is optional. If a value isn't passed to `y`, the default value is `25`.

TypeScript

```
function addNumbers (x: number, y = 25): number {
   return x + y;
}

addNumbers(1, 2);  // Returns 3
addNumbers(1);     // Returns 26
```

Rest Parameters

If you want to work with multiple parameters as a group (for example, passing them in an array). Or, if you don't know how many parameters a function will ultimately take. You can use rest parameters. Rest parameters are treated as a boundless number of optional parameters. You may leave them off or have as many as you want.

This example has one required parameter and an optional parameter called `restOfNumbers` that can accept any number of additional numbers. The ellipsis (`...`) before `restOfNumbers` tells the compiler to build an array of the arguments passed to the function and assigns the name that follows to it so you can use it in your function.

TypeScript

```
let addAllNumbers = (firstNumber: number, ...restOfNumbers: number[]): number => {
   let total: number =  firstNumber;
   for(let counter = 0; counter < restOfNumbers.length; counter++) {
      if(isNaN(restOfNumbers[counter])){
         continue;
      }
      total += Number(restOfNumbers[counter]);
   }
   return total;
}
```

The function can now accept one or more values and return the result.

TypeScript

```
addAllNumbers(1, 2, 3, 4, 5, 6, 7);  // returns 28
addAllNumbers(2);                    // returns 2
addAllNumbers(2, 3, "three");        // flags error due to data type at design
time, returns 5
```

Deconstructed object parameters

Function parameters are positional and must be passed in the order in which they're defined in the function. Positional parameters can make your code less-readable when calling a function with multiple parameters that are optional or the same data type.

To enable named parameters, you can use a technique called deconstructed object parameters. This technique enables you to use an interface to defined named, rather than positional, parameters in your functions.

The following example defines an interface called `Message` that defines two properties. In the `displayMessage` function, the `Message` object is passed as a parameter, providing access to the properties as if they're regular parameters.

TypeScript

```
interface Message {
   text: string;
   sender: string;
}

function displayMessage({text, sender}: Message) {
    console.log(`Message from ${sender}: ${text}`);
}

displayMessage({sender: 'Christopher', text: 'hello, world'});
```

Next unit: Exercise - Fun with parameters

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆